Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Urban Marovt

Zasnova skalabilnega in visoko dostopnega prehoda za upravljanje programskih vmesnikov

diplomsko delo

na interdisciplinarnem univerzitetnem študiju
računalništva in matematike

Mentor: red. prof. dr. Matjaž Branko Jurič

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preučite področje programskih vmesnikov in njihovega upravljanja ter identificirajte ključne funkcionalnosti sistemov za upravljanje programskih vmesnikov. Analizirajte prehode, opišite njihovo delovanje in ključne lastnosti. Zasnujte arhitekturni načrt lastnega prehoda in ga implementirajte z uporabo Node.js. Pri tem zasledujte cilje visoke odzivnosti, skalabilnost in elastičnosti. Implementiran prehod namestite z uporabo vsebnikov Docker in simulirajte njegovo elastičnost in skalabilnost z uporabo ogrodja Kubernetes. Skalabilnost prehoda testirajte na določeni množici zahtevkov in ovrednotite rezultate.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Urban Marovt, z vpisno številko 63120287, sem avtor zaključnega dela z naslovom:

Zasnova skalabilnega in visoko dostopnega prehoda za upravljanje programskih vmesnikov

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom prof. dr. Matjaža B. Juriča;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 5. september 2016 Podpis študenta/-ke:

Zahvaljujem se mentorju red. prof. dr. Matjažu Branku Juriču in Juretu Tuti za podporo in vsa mnenja pri pisanju diplomske naloge. Zahvalil bi se tudi moji družini in Petri za vso podporo v času mojega študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	Programski vmesnik
KPI	Key Performance Indicator	Ključni kazalnik uspešnosti
SLA	Service-Level Agreement	Sporazum o zagotavljanju storitev
SOA	Service Oriented Architecture	Storitveno usmerjena arhitektura
DOM	Document Object Model	Objektni model dokumenta
REST	Representational State Transfer	Predstavitveni prenos stanja
SOAP	Simple Object Access Protocol	Protokol za dostop do objektov
XML	Extensible Markup Language	Razširljiv označevalni jezik
XML-RPC	XML Remote Procedure Call	Oddaljen klic procedur z uporabo XML
JSON	JavaScript Object Notation	Javascript objektni zapis
BSON	Binary JavaScript Object Notation	Binarni Javascript objektni zapis
DoS	Denial of Service	Napad za zavrnitev storitve
SQL	Structured Query Language	Strukturiran povpraševalni jezik

Povzetek

Kot odgovor na povečano uporabo programskih vmesnikov za izpostavljanje podatkov in programske logike, se je pojavilo področje upravljanja programskih vmesnikov. Z naraščanjem količine podatkov, predvsem števila zahtevkov, pa se danes pojavlja potreba po elastični, skalabilni in zanesljivi arhitekturi takšnih sistemov. V diplomskem delu smo opisali arhitekturno zasnovo skalabilnega in visoko dostopnega sistema, ki izvaja funkcionalnosti upravljanja programskih vmesnikov. Razvili smo prehod in pri implementaciji uporabili dogodkovno usmerjeno Javascript programsko ogrodje Node.js. S pomočjo orkestracijske tehnologije Kubernetes smo implementirano komponento, ovito v vsebnike Docker, postavili v oblaku in opravili teste učinkovitosti ter obremenitvena testiranja, ki smo jih predstavili v rezultatih diplomskega dela. Pokazali smo, da se odzivni čas zahtevkov zaradi podaljšane poti skozi prehod povprečno podaljša za 10 milisekund, hkrati pa lahko izpostavljen prehod nemoteno deluje pri obremenitvi 1000 zahtevkov na sekundo.

Ključne besede: programski vmesnik, prehod, upravljanje programskih vmesnikov, skalabilnost, dostopnost.

Abstract

As a consequence of the increasing interest for application programming interfaces (APIs) the API management field has rapidly gained importance. With the quick rise in the amount of data on the Internet and increased amount of requests, there is a great need for elastic, scalable and reliable infrastructure of such systems. In our thesis we discuss an approach to the described problems and propose an architecture of an API management system. We developed API gateway, which we implemented in Node.js server-side Javascript framework. We deployed the implemented component wrapped in Docker containers using the Kubernetes orchestration tool. In the results section of the thesis we discuss the load and effectiveness testing of the deployed system. We proved that as a consequence of the extended path through the implemented gateway, the response time of requests has increased in average for 10 milliseconds and that the gateway can handle the load of 1000 request per second.

Keywords: application programming interface, gateway, API management, scalability, availability.

Poglavje 1

Uvod

Zaradi hitre rasti količine podatkov na spletu je danes velik poudarek pri razvoju spletnih tehnologij na skalabilnosti in šibki sklopljenosti spletnih sistemov ter aplikacij. Zelo pomemben vidik tega področja je način izpostavljanja programske logike in podatkov, kjer so se v zadnjih nekaj letih močno uveljavili programski vmesniki (angl. application programming interface, API), do katerih dostopamo preko končnih točk (angl. end points). Zaradi večje pomembnosti izpostavitve programske logike in podatkov pa se je v zadnjih letih močno razvilo tudi področje upravljanja in vodenja programskih vmesnikov. To zajema predvsem varnost izpostavljanja pogramskih vmesnikov, način izpostavljanja, analizo in nadzor posameznih končnih točk ter enostavno razširljivost sistema. Pojavilo se je veliko število poslovnih rešitev za integracijo programskih vmesnikov, ki omogočajo uporabo že izdelanih sistemov za prej naštete aktivnosti. Po navedbah raziskave [3] iz leta 2015 najboljše rešitve predstavljajo SOA Software, Microsoft API Management, Apigee Edge in CA Layer 7 API Management.

Za vsak sistem je pomembno, da ga lahko enostavno integriramo v že implementiran strežniški vmesnik. Ponavadi imajo ponudniki takšnih rešitev dve možnosti: sistem direktno integriran v poslovno logiko, ki stoji za izpostavljenimi končnimi točkami ali sistem, ki stoji pred končnimi točkami in deluje kot samostojna enota v obliki prehoda (angl. gateway). O takšnem

načinu, ki vključuje samostojno prehodno enoto ali obratni namestniški strežnik (angl. reverse proxy server), govori tudi koncept storitveno usmerjene arhitekture (angl. service oriented architecture, SOA), pri kateri je poudarek ravno na enostavni skalabilnosti in šibki sklopljenosti posameznih komponent.

Pri takšni arhitekturni zasnovi se lahko ob preveliki obremenitvi pojavijo težave s hitrostjo in nestabilnostjo sistema. Potrebno je zasnovati sistem, ki bo imel čim manj kritičnih točk, ki bi lahko povzročile nedelovanje celotnega sistema, in bo hkrati deloval hitro kljub temu, da bo poleg osnovnih nalog skrbel še za kopico ostalih. Zgoraj omenjene rešitve ne dajejo poudarka na problematiko elastičnosti sistemov, prilagajanje sistemov velikim obremenitvam in dovzetnosti sistemov za napake. Cilj diplomskega dela je zasnovati primerno arhitekturo prehoda, implementiranega v ogrodju Node.js, ki bo pokrival določene naloge upravljanja programskih vmesnikov. Cilj je zasnovati skalabilen in visoko dostopen prehod, hkrati pa želimo preveriti primernost izbrane tehnologije za razvoj takšnega sistema.

V diplomskem delu se bomo najprej osredotočili na področje upravljanja programskih vmesnikov. V tretjem in četrtem poglavju bomo kot možno rešitev za upravljanje programskih vmesnikov izpostavili komponento prehoda in predstavili našo implementacijo določenih funkcionalnosti. V petem poglavju se bomo osredotočili na skalabilnost in visoko dostopnost arhitekturne zasnove in načine izpostavitve sistema s pomočjo orkestracijske tehnologije Kubernetes. S testi učinkovitosti in obremenitvenimi testi bomo na koncu preverili hitrost obdelave zahtevkov in skalabilnost izpostavljenega sistema, hkrati pa bomo preverili tudi primernost izbrane tehnologije Node.js.

Poglavje 2

Upravljanje programskih vmesnikov

Pri vsakem sistemu se slej kot prej pojavi potreba po varnosti, nadzoru in spremljanju delovanja. S skupnim izrazom to imenujemo upravljanje programskih vmesnikov (angl. API management). Bralec se bo v tem poglavju spoznal z osnovnimi vidiki in funkcionalnostmi upravljanja programskih vmesnikov.

2.1 Avtentikacija aplikacij in uporabnikov

Ena glavnih nalog upravljanja programskih vmesnikov je skrb za varnost in avtentikacijo uporabnikov. Avtentikacija je pomembna za skoraj vse vidike uporabe vmesnikov. Minimalno avtentikacijo dosežemo z uporabo identifikacijskega ključa, ki ga pošilja aplikacija ob vsakem zahtevku. Tako lahko identificiramo aplikacijo, s katero uporabniki dostopajo do končnih točk programskega vmesnika. Ob zahtevku lahko identificiramo še vsakega uporabnika posebej in mu ob prvem zahtevku podamo ključ, s katerim lahko dostopa do končnih točk vmesnika za določen čas.

2.2 Varnost sistema

Veliko pozornosti je tako pri zasebni kot tudi pri javni vzpostavitvi potrebno nameniti varnosti. Pri javni objavi sistema je potrebno paziti predvsem na napade zavrnitve sistema (angl. denial of service attack, DoS), katerih cilj je doseči začasno preobremenjenost sistema in posledično nedelovanje. Klasična obramba pred temi napadi je nadzor pretoka zahtevkov (angl. bandwidth throttling) [19], kar pomeni, da omejimo maksimalno število le-teh v določenem časovnem obdobju. Pri tem se lahko odločimo za nadzor glede na posameznega uporabnika ali celotno aplikacijo, lahko pa nadzor ločimo po posameznih končnih točkah [20]. Pri zasebnih programskih vmesnikih lahko naletimo na enako težavo, poskrbeti pa moramo tudi za dober avtentikacijski sistem, ki preprečuje nezaželjene dostope.

2.3 Nadzor pretoka in uporabe

Želja večine ponudnikov je imeti dober nadzor nad uporabo njihovih programskih vmesnikov, zato se informacije o vsakem zahtevku shranijo. Pri nadzoru uporabe gre za ponudnikovo izpostavitev različnih načinov porabe (angl. plan), ki na koncu definirajo sporazum o zagotavljanju storitev (angl. service level agreement, SLA). To je dokument ali pogodba med ponudnikom in odjemalcem, ki definira njuno sodelovanje v obliki obsega, kvalitete in odgovornosti. Za zagotavljanje in dosledno izvajanje pogodbe je za ponudnika zelo pomemben nadzor uporabnikovega delovanja. Za potrebe nadzora pretoka in uporabe je ponavadi definirano največje število dostopov v določenem časovnem obdobju, ponudnik pa se zaveže za kvaliteto in zanesljivost delovanja programskega vmesnika.

2.4 Preoblikovanje in preusmerjanje zahtevkov

Vsak programski vmesnik sprejema podatke predpisane oblike in po določenem protokolu. Velikokrat so podatki v strukturi imenovani Razširljiv označevalni jezik (angl. Extensible markup language, XML). V zadnjih letih pa se uveljavlja struktura Javascript objektni zapis (angl. Javascript object notation, JSON). Sama struktura je ponavadi povezana tudi s komunikacijskim protokolom, ki ga uporablja izpostavljen programski vmesnik. Najpogosteje uporabljeni so XML-RPC, SOAP in REST. Zadnjega smo tudi mi uporabili pri sprejemanju zahtevkov. Seveda je izbira protokola odvisna od različnih dejavnikov. V primeru javne izpostavitve programskega vmesnika je velikokrat pametno zadostiti več potrebam. V ta namen se na prehodu implementira preoblikovanje zahtevkov med različnimi strukturami in različnimi komunikacijskimi protokoli.

2.5 Preusmerjanje zahtevkov

Sistem za upravljanje programskih vmesnikov je ponavadi ločena strežniška enota, ki izpostavlja lastne končne točke in ima svoj naslov. Ker se pot zahtevkov tam ne konča, je potrebno implementirati algoritem za njihovo preusmerjanje. To je odvisno od števila in oblike sistemov ter strežnikov, ki stojijo za prehodom. Če je izpostavljen en programski vmesnik, je to preusmerjanje enostavno, v nasprotnem primeru pa se zahtevana funkcionalnost lahko implementira na dva načina: preko unikatnega ključa, ki ga uporabnik pošlje v glavi zahtevka, ali pa prehod izpostavi več končnih točk, preko katerih se nato določi končni strežnik prejetega zahtevka.

2.6 Spremljanje zahtevkov

Vsakega ponudnika programskega vmesnika zanima, kako se obnašajo uporabniki, kje se pojavlja največ težav in kako uspešno deluje sistem. Temu pravimo spremljanje (angl. monitoring) in deluje na podlagi prej omenjenega shranjevanja informacij o vsakem prejetem zahtevku. Zaradi mnogih karakteristik programskih vmesnikov lahko ponudnik nadzoruje več vidikov, kot so na primer število klicev, priljubljenost posameznih končnih točk, v primeru javne objave število programerjev, ki aktivno uporabljajo ta vmesnik, hitrost, dostopnost in kakovost delovanja sistema [11]. Kaj dejansko merimo in kaj nas mora zanimati, je odvisno od same narave sistema, vse to skupaj pa imenujemo ključni kazalniki uspešnosti (angl. key performance indicator, KPI) [18]. To je ena najpomembejših nalog upravljanja s programskimi vmesniki, saj ponudnik izve veliko o samih uporabnikih in delovanju njegovega sistema.

2.7 Druge funkcionalnosti

V predhodnjih poglavjih so opisani vidiki upravljanja s programskimi vmesniki, ki so za naše delo pomembni. Področje programskih vmesnikov je veliko širše od opisanih funkcionalnosti, zato bomo na kratko izpostavili še druge pomembnejše vidike [26]. Zelo pomemben del vodenja programskih vmesnikov je njihova objava na spletu. Sem spada hitro odkrivanje objavljenega vmesnika in prevzem podatkov s pomočjo dobro pripravljene dokumentacije in začetnega vodiča. V zadnjem času se vedno bolj uveljavlja dokumentacija po posameznih končnih točkah, kjer so zbrani podatki o primeru zahtevka, obliki odgovora, možnih prejetih napakah ter primeri kode v najbolj standardnih jezikih. Večkrat srečamo tudi interaktivno dokumentacijo, kjer lahko dejansko izvedemo klice in preizkusimo odzivnost API-ja brez pisanja lastne kode. Hkrati je pomembno imeti dober način upravljanja z različicami in učinkovito storitev za pomoč tako uporabnikom kot tudi razvijalcem.

Poglavje 3

Prehod

Cilj tega poglavja je pregled tehničnih in arhitekturnih značilnosti implementacije prehoda. Na začetku poglavja se bralec spozna z osnovami prehoda, njegove arhitekturne umestitve in prednostmi ter slabostmi njegove uporabe. V drugem delu poglavja so podrobneje opisane funkcionalnosti prehoda glede na pot zahtevka skozi prehod.

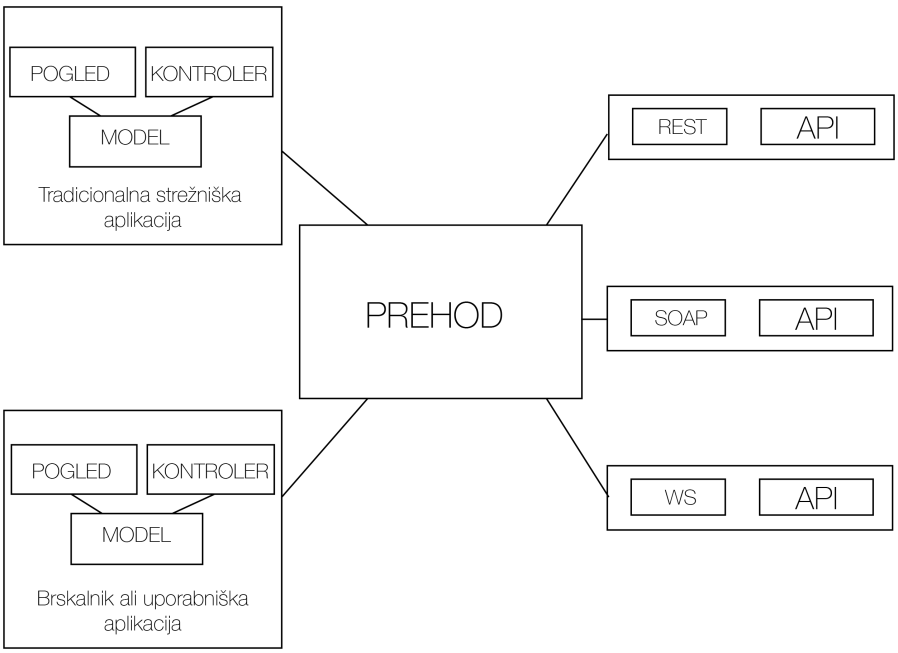
3.1 Osnove lastnosti prehoda

Prehod je enota, ki jo navadno označujemo kot skupno vstopno točko v strežniški sistem. V osnovi pa je prehod osrednja točka upravljanja programskih vmesnikov, saj se na njem izvajajo vsi procesi, ki smo jih opisali v poglavju 2. Naložimo ga lahko na skupen strežnik s programskim vmesnikom, velikokrat pa ju tudi fizično ločimo. Zelo pomembno je, da se odzivni čas sistema (čas med poslanim zahtevkom in prejetim odgovorom) zaradi funkcionalnosti prehoda ne poveča pretirano in da normalno deluje tudi ob velikih obremenitvah.

3.1.1 Postavitev prehoda v sistemu

Prehod stoji pred izpostavljenimi programskimi vmesniki in je skupna vstopna točka v sistem. Na sliki 3.1 je prikazan osnovni arhitekturni koncept ob

uporabi prehoda. Vidimo, da imajo lahko različni programski vmesniki skupen prehod, nanj pa lahko dostopamo preko različnih protokolov in sistemov.



Slika 3.1: Preprosta arhitekturna shema z uporabo prehoda.

3.1.2 Prednosti in slabosti uporabe prehoda

Osnovna prednost prehoda kot ločene enote v strežniškem sistemu je neodvisnost od ostalih komponent, kar posledično prinese šibko sklopljenost sistema. Poleg tega lahko izpostavimo še druge pozitivne lastnosti:

- deluje kot namestniški obratni strežnik, kar pomeni, da uporabnik oz. odjemalec ne vidi implementacije poslovne logike v ozadju, ampak je ta skrita za končnimi točkami prehoda,
- odjemalcem ni potrebno skrbeti za lokacijo programskega vmesnika, ampak le za lokacijo prehoda,

- zaradi možnosti preoblikovanja zahtevkov lahko preko prehoda uporabniku ponudimo optimalno obliko odgovorov in vrste komunikacije, neodvisno od implementacije programskega vmesnika,
- zmanjša se lahko število zahtevkov, če le-te združujemo na prehodu in tako poenostavimo proces pridobivanja podatkov v odjemalskih aplikacijah.

Neustrezna implementacija prehoda lahko izniči vse naštete prednosti, zato je zelo pomembno preveriti hitrost delovanja sistemov in se osredotočiti na dobro arhitekturno zasnovo. Pri tem pa seveda nikoli ne smemo pozabiti na negativne lastnosti, ki jih prinese uporaba prehoda:

- potreba po razvoju, namestitvi in vzdrževanje dodatne komponente sistema,
- podaljšan odzivni čas zahtevkov,
- povečanje števila skokov v omrežju,
- možnost, da prehod postane ozko grlo sistema.

3.1.3 Tipična pot zahtevka skozi prehod

Ob prejetem zahtevku se nad njim znotraj sistema prehoda izvede kopica operacij, zato lahko njegovo pot skozi prehod razdelimo na štiri dele [17]. Sprejemanje zahtevka se zgodi v transportnem nivoju prehoda. Zatem pride zahtevek v oddaljen nivo, kjer se izvede večina procesov, ki smo jih opisali v poglavju 2. Na koncu sledi predaja ustreznih podatkov preko modelnega nivoja do povezovalnega, ki na koncu izvede ustrezne klice na oddaljene programske vmesnike. Ta koncept je predstavljen na sliki 3.2.

Pot zahtevka skozi prehod pa lahko orišemo tudi iz vidika, kaj vse se dogaja s samim zahtevkom. Pri tem lahko izpostavimo osem glavnih korakov:

1. zahtevek za pridobivanje avtentikacijskega žetona,

2. dejanski zahtevek je skupaj s pridobljenim žetonom poslan na prehod,
3. prehod preveri veljavnost žetona in možnost nadaljnje obdelave zahtevka,
4. izvedejo se vsi procesi, ki so potrebni za shranjevanje podatkov o zahtevku,
5. izvedejo se vsi procesi, ki nadzorujejo skladnost zahtevka s SLA med pošiljateljem in ponudnikom,
6. prehod se na podlagi zahtevka in njegovih parametrov odloči, ali bo le-ta obdelan lokalno ali pa bo poslan naprej,
7. zahtevek je iz prehoda poslan naprej na ustrezen programski vmesnik,
8. prehod prejme odgovor z vmesnika, ponovno posodobi podatke o zahtevku in ga pošlje odjemalcu.

Slika 3.2: Prehod razdeljen po nivojih [17].

Poglavje 4

Implementacija prehoda

To poglavje opisuje značilnosti naše implementacije prehoda. Na začetku smo opisali tehnične lastnosti uporabljenih tehnologij in razloge za njihovo uporabo. V drugem delu poglavja pa smo se osredotočili na razlago same implementacije sistema.

4.1 Tehnične lastnosti implementacije

Kot smo že v uvodu napisali, smo za razvojno okolje izbrali odprtokodno ogrodje Node.js, ki izvaja skripte, napisane v programskem jeziku Javascript. V tehničnem pregledu bomo opisali uporabljene module Node.js in sisteme za shranjevanje podatkov.

4.1.1 Javascript

Javascript, ki je standardiziran kot programski jezik ECMAScript, je eden najbolj razširjenih programskih jezikov, ki je bil do nedavnega večinoma implementiran zgolj znotraj spletnih brskalnikov, kjer med drugim omogoča dostopanje do elementov objektnega modela dokumenta (angl. document object model, DOM), interakcijo z uporabnikom in izvajanje asinhronih klicev. Javascript je dinamično in šibko tipiziran programski jezik, kjer je vsak element predstavljen kot objekt Javascript, njegovo osnovo pa predstavlja

tip Prototype [8], ki ga imenujemo prosto razredni objektni sistem. Prototipno dedovanje je ena pomembnejših značilnosti programskega jezika Javascript in je zelo močno orodje. Omogoča, da lahko vsak objekt deduje vse parametre poljubnega objekta, kar ni značilno za ostale programske jezike. Objekti v Javascriptu so prosto razredni, kar pomeni, da lahko vsakemu objektu pripišemo poljuben parameter s poljubno vrednostjo. Tako lahko enostavno predstavimo strukturo drevesa ali grafa. Javascript obenem razvijalcu omogoča tudi različne pristope programiranja kot so funkcijsko, objektno in imperativno.

Po drugi strani se je pomembno zavedati določenih negativnih lastnosti Javascripta. Ena glavnih je, da se za povezavo uporabljajo globalne spremenljivke, ki so zbrane v imenskem prostoru globalnega objekta. Globalne spremenljivke, ki nikoli niso bile dobra izbira, so v Javascriptu osnova. Problem globalnih spremenljivk se lahko s pomočjo določenih korakov uspešno zaobide, a je v osnovi vseeno prisoten. Zaradi svoje dinamičnosti in šibke tipiziranosti mora biti razvojnik dovolj dosleden, saj lahko programska koda hitro postane nepregledna in posledično neuporabna.

Že v zgodnji dobi interneta so se aplikacije tudi na strežniški strani razvijale v programskem jeziku Javascript. To je podpiral Netscape Enterprise Server, vendar je njegov razvoj omejevala takratna počasnost izvajanja programskega jezika Javascript, zato so prevladale druge alternative, kot sta programski jezik PHP in razvojno orodje ASP.NET. Zaradi razvoja hitrih in zelo dovršenih virtualnih naprav Javascript in na njih baziranih okoljih, ki so se pojavili v zadnjih letih, pa se je znova začel uporabljati tudi na strežniški strani. Razvilo se je kar nekaj strežniških ogrodij, med katerimi pa najbolj izstopa ravno Node.js.

4.1.2 Node.js

Razvoj ogrodja Node.js se je začel leta 2009 in je v zadnjih letih pridobil na popularnosti kot rešitev za razvoj strežniške programske opreme. Ogrodje Node.js sledi dogodkovni in asinhroni arihitekturi, kar omogoča lažjo skala-

bilnost in preprostost sistemov. Aplikacija Node.js v osnovi teče zgolj na eni niti, zato ni primeren za izvajanje procesorsko težkih operacij, ampak se večinoma uporablja v primerih, ko se pojavi potreba po hitrem in propustnem sistemu. Za sproščanje uporabljene niti skrbi dogodkovna arhitektura, ki vse operacije razdeli v dve funkciji. Prva kliče oddaljeno komponento, kot sta podatkovna zbirka ali oddaljen strežnik, druga pa obdela vrnjen rezultat. Vmes nit ni blokirana in lahko obdeluje druge zahtevke.

Čeprav takšen pristop ne izkorišča vseh prednosti večjedrnih procesorjev, se vedno bolj uveljavlja kot dobra alternativa večnitni arhitekturi (angl. multithreading architecture), ki jo kot primer uporablja programski jezik Java. Avtorja članka [15] iz leta 2010 izpostavita, da dogodkovna arhitektura programerju prepušča več svobode pri kontroli vrstnega reda izvajanja procesov in nima težav pri souporabi virov. To težavo izpostavijo tudi avtorji sistema SEDA [16], ki hkrati poudarijo počasnost operacijskih sistemov pri razporejanju procesov ob veliki obremenitvi. Prednost eno-nitne arhitekture je tudi neomejeno število sprejemanja zahtevkov. Nasprotno lahko pri večnitni arhitekturi velika količina hkratnih povezav povzroči pomanjkanje prostora na pomnilniku.

Prednost ogrodja Node.js in ostalih strežniških okolij Javascript je v tem, da lahko celotno spletno aplikacijo sedaj napišemo v enem skupnem programskem jeziku. Okoli ogrodja Node.js se je razvil zelo velik ekosistem s ponudbo več tisoč modulov, ki jih lahko vključimo v svoje aplikacije. Te ponavadi pridobivamo s pomočjo paketnega upravljalca (angl. package manager) ogrodja Node.js Npm. Tudi mi smo pri razvoju uporabili nekaj komponent ogrodja Node.js, ki so nam olajšale delo:

- pri izpostavljanju končnih točk (uporabljen modul Express),
- pri posredovanju zahtevkov (uporabljen modul Request),
- pri dostopanju do podatkovih baz (uporabljena modula Redis-Sentinel in Mongodb).

4.1.3 Redis

Redis je odprtokodna podatkovna zbirka, kjer so podatki shranjeni v obliki ključ-vrednost (angl. key-value) in spada v skupino nerelacijskih podatkovnih zbirk. Zbirka večino podatkov hrani v pomnilniku in kot navaja R. Cattell [7] v svoji raziskavi iz leta 2011, posledično velja za izredno hitro podatkovno zbirko. Strežnik Redis vseskozi usklajuje, kateri podatki so v pomnilniku in kateri so shranjeni v datotečnem sistemu. Podatkovno zbirko Redis smo uporabili za shranjevanje vrednosti, ki si jih strežniki gruče prehodov med sabo delijo. V primerjavi z ostalimi podatkovnimi zbirkami tipa ključ-vrednost, kot sta Project Voldemort in Membase, prednost predstavlja ravno njegova hitrost, za nas pa je bilo pomembno tudi dejstvo, da omogoča uporabo transakcij. Tako kot strežnik Node.js tudi Redis deluje na enem procesorju v zgolj eni niti, zato simultana obdelava podatkov ni mogoča. V novejših verziji podatkovna zbirka Redis podpira posebno datoteko, na podlagi katere lahko uporabnik določa, kateri podatki bodo shranjeni na strežniku in kateri v pomnilniku.

Podatkovna zbirka Redis podpira podvajanje podatkov med strežniki na podlagi razmerja gospodar-suženj (angl. master-slave) s pomočjo funkcionalnosti Redis Cluster. Tako lahko ustvarimo drevo strežnikov, ki vsebujejo enake podatke, na njih pa lahko izvajamo vse operacije razen zapisovanja v podatkovno zbirko, ki ga lahko izvajamo zgolj na korenskem strežniku drevesa. Funkcionalnost Redis Sentinel z dodatnimi strežniki poskbi za visoko dostopnost strežnikov podatkovne zbirke Redis, omogoča spremljanje delovanja strežnikov in obveščanje razvijalcev v primeru nedelovanja sistema. Podvajanje je pri skalabilnosti podatkovnih zbirk pomembno iz vidika zagotavljanja visoke dostopnosti in omogočanja delovanja sistema v primeru izpada enega izmed strežnikov.

4.1.4 MongoDB

Pri implementaciji smo potrebovali dodatno podatkovno strukturo za beleženje informacij o delovanju sistema in prejetih zahtevkih ter informacije o registriranih aplikacijah v ozadju. K. Ma in R. Sun [10] sta v članku iz leta 2013 izpostavila težave relacijskih podatkovnih zbirk ob veliki obremenitvi in pri povpraševanju nad ogromno količino podatkov. Kot rešitev sta navedla nerelacijske podatkovne zbirke, ki pri omenjenih funkcionalnostih delujejo hitreje in bolje.

Posledično smo tudi mi uporabili nerelacijsko podatkovno zbirko, podrobneje pa smo analizirali možnost uporabo zbirk MongoDB in CouchDB. Podatkovna zbirka CouchDB je dokumentna podatkovna zbirka, v kateri so podatki shranjeni v formatu JSON. Do shranjenih podatkov lahko dostopamo preko REST HTTP klicev na programski vmesnik podatkovne zbirke.

Med podatkovnima zbirkama MongoDB in CouchDB je veliko podobnosti, vendar je podatkovna zbirka MongoDB bolj primerna za naše potrebe, saj ima določene prednosti pri grupiranju in horizontalnemu skaliranju. V članku [7] avtorji izpostavijo avtomatizirano podvajanje podatkov med strežniki, kar omogoča, da bomo imeli pripravljene varnostne kopije. Omogoča tudi dinamično povpraševanje, ki je zelo dobrodošlo pri implementaciji uporabniškega vmesnika za potrebe spremljanja delovanja programskega vmesnika, predvsem pa ima boljšo implementacijo indeksiranja, ki omogoča hitrejše povpraševanje.

Podatkovna zbirka MongoDB predstavlja dokumentno podatkovno strukturo. Vsak element je shranjen v obliki, podobni notaciji JSON, ki jo pri zbirki MongoDB imenujejo Binarni Javascript objektni zapis (angl. binary Javascript object notation, BSON). Ker se polja zahtevkov med seboj razlikujejo, nam dokumentna struktura omogoča, da lahko vse kljub temu shranimo v enotno zbirko podatkov.

Pomembna lastnost podatkovne zbirke MongoDB je poleg prej omenjene dokumentne strukture predvsem način povpraševanja. Podatkovna zbirka MongoDB omogoča ad hoc povpraševanje po polju in razmaku vrednosti ter

povpraševanja z uporabo regularnih izrazov. Prednost pri povpraševanju pa ima tudi orodje Node.js, saj lahko podatke podatkovnih zbirk preiskujemo s pisanjem programske kode Javascript in ni potrebe po uporabi dodatnega povpraševalnega jezika.

Za hitrejše delovanje je omogočeno primarno in sekundarno indeksiranje ravno tako kot v relacijskih podatkovnih zbirkah, kar je zelo pomembno za naš sistem, saj se bo količina podatkov hitro povečevala. Enako kot podatkovna zbirka Redis tudi zbirka MongoDB omogoča podvajanje podatkov med strežniki, kar nam je omogočilo, da smo poskrbeli za visoko odzivnost sistema in podatkovno drobljenje (angl. sharding), kot imenujemo metodo horizontalnega skaliranja pri podatkovni zbirki MongoDB.

4.2 Implementacija funkcionalnosti prehoda

Na začetku naloge smo opisali, čemu služi prehod in zakaj bi ga nekdo želel vključiti v svoj sistem. Skupek vseh vidikov upravljanja programskih vmesnikov je zelo kompleksen in redko kateri sistem pokriva vse naštete vidike. Zato smo se tudi mi osredotočili le na nekaj vidikov problematike upravljanja programskih vmesnikov:

- preusmerjanje zahtevkov,
- nadzor pretoka in uporabe,
- spremljanje,
- preoblikovanje zahtevkov.

Ker je namen diplomske naloge preveriti primernost arhitekturne zasnove in orodja Node.js kot platforme za implementacijo prehoda, so opisane funkcionalnosti dovolj, da lahko odgovorimo na zastavljeni vprašanji.

4.2.1 Preusmerjanje zahtevkov

Za našim prehodom lahko stoji poljuben programski vmesnik, zato je potrebno vsakemu ob registraciji določiti unikaten ključ, na podlagi katerega bomo prejeti zahtevek ustrezno preusmerili. To funkcionalnost v osnovi implementiramo na obratnem namestniškem strežniku.

Pri implementaciji preusmerjanja zahtevkov imamo več možnosti. Najenostavnejša je, da od uporabnika zahtevamo, da v glavo zahtevka vključi dodaten atribut, ki nam bo služil kot identifikacijski ključ posameznega programskega vmesnika. Pri tej rešitvi nimamo težav z razčlenjevanjem naslova, vendar pa se težava pojavi takrat, ko bi enak atribut zahteval tudi programski vmesnik, ki stoji za prehodom.

Druga rešitev je, da je poleg naslova tudi prvi parameter v poti naslova URL, povezan z delovanjem prehoda. Ta rešitev nima težav ujemanja z izpostavljenim programskim vmesnikom, vendar pa ni najbolj ustrezna, saj pri tem med seboj pomešamo parametre programskega vmesnika in prehoda. Hkrati je potrebno za pridobitev identifikacijskega ključa razčleniti parametre poti prejetega zahtevka in odvečnega nato eliminirati.

Najustreznejša implementacija preusmerjanja zahtevkov je, da indentifikacijski niz vključimo pred naslov, kot da bi naš strežnik sprejemal zahtevke na določeno poddomeno. Vseeno je potrebno za pridobitev idenfikacijskega ključa razčleniti naslov, vendar v tem primeru ne potrebujemo dodatnih sprememb, le naslov je potrebno prepisati in zahtevek posredovati naprej. Hkrati je ta rešitev tudi najbolj čitljiva in pregledna.

Pri implementaciji prehoda smo uporabili zadnjega izmed opisanih pristopov. Glede na identifikacijski ključ, ki ga pridobimo, nato izvedemo poizvedbo na podatkovno zbirko, ki vsebuje podatek o naslovu ustreznega programskega vmesnika, kamor kasneje preusmerimo zahtevek.

4.2.2 Nadzor pretoka in uporabe

Nadzor pretoka lahko tako kot preusmerjanje zahtevkov implementiramo na različne načine [20]. Namen je pri vseh implementacijah enak: omejiti zgornje število klicev v določenem časovnem obdobju. Kaj se zgodi, ko uporabnik doseže zgornjo mejo, je odvisno od implementacije. Zahtevke lahko začnemo zavračati ali pa uporabimo pristop mehkejše implementacije in odjemalcu pošiljamo le opozorila. Lahko povečamo odzivni čas zahtevkov ali pa celo vsak naslednji prejeti zahtevek dodatno zaračunamo.

Implementacija na podlagi števca

Prvi način implementacije je na podlagi števca zahtevkov. V osnovi deluje tako, da čas razdelimo v razdelke enake dolžine, vmes pa spremljamo prihajajoče zahtevke. Ko pride zahtevek, preverimo stanje števca. Najprej pogledamo, če je časovno obdobje poteklo in lahko števec ponastavimo, zatem pa preverimo stanje števca. Če še nismo prekoračili zgornje meje, števec povečamo in zahtevek posredujemo naprej, drugače pa zahtevek zavržemo.

Implementacija opisanega pristopa je enostavna, saj zadostuje že navaden slovar, vendar ima tak pristop eno težavo. Predstavljajmo si, da imamo dva zaporedna časovna razdelka. Pri prvem v drugi polovici pošljemo vse zahtevke, pri drugem pa v prvi polovici. Tako smo v enem časovnem obdobju poslali dvojno število zahtevkov in na vse dobili odgovor. Z večanjem časovnega okvirja, se ta težava povečuje, zato je takrat bolje izbrati drugačen pristop.

Implementacija na podlagi vrste

Drugi način implementacije pa je na podlagi vrste. Ta deluje tako, da zahtevke shranjujemo v vrsto. Ob prejetem zahtevku najprej iz vrste izločimo zahtevke, ki so bili prejeti pred časovno omejitvijo, zatem pa preštejemo število zahtevkov v vrsti in če je teh več, kot je zgornja meja, zahtevek zavrnemo, v nasprotnem primeru pa ga dodamo v vrsto in ga posredujemo

naprej.

Razlaga izbrane implementacije

Zaradi naštetih prednosti smo se odločili za implementacijo na podlagi vrste. Ker moramo zaradi horizontalno skalabilne arhitekture poskrbeti, da imajo vse enote gruče (angl. cluster) vedno na razpolago aktualno stanje, smo implementirali sistem brez stanja in vse spremenljivke shranili v podatkovno zbirko Redis. Operacije preverjanja pretoka moramo izvajati kot celoto znotraj transakcij, saj ne smemo dovoliti, da se med preverjanjem pretoka spremeni stanje opazovane spremenljivke v podatkovni zbirki.

Izbrati smo morali strukturo, ki bo imela hitro vstavljanje, štetje elementov in brisanje elementov. Za implementacijo smo izbrali urejeno množico (angl. sorted set) implementirano v podatkovni zbirki Redis. Vsakemu vstavljenemu elementu poleg vrednosti dodamo vrednost po kateri so urejeni elementi v množici. V našem primeru to vrednost nastavimo na čas, ko smo prejeli zahtevek. Funkcija dodajanje elementa v strukturo ZADD ima časovno zahtevnost $O(\log n)$, kjer je n trenutno število elementov v množici. Število vseh elementov v množici dobimo s klicem funkcije ZRANK, ki ima konstantno časovno zahtevnost $O(1)$. Uporabljamo pa tudi funkcijo ZREMRANGEBYSCORE, ki odstrani vse elemente z nižjo vrednostjo od podane in ima časovno zahtevnost $O(\log n + m)$, kjer je m število odstranjenih elementov. Vsak element bo enkrat dodan v množico in enkrat izbrisan iz množice, torej spremenljivka m ne vpliva na skupno časovno zahtevnost.

Opazimo, da bi bila boljša struktura glede na časovne zahtevnosti operacij vrsta (angl. queue), kjer vstavljanje in preštevanje elementov opravimo v konstantnem času (ob predpostavki, da imamo število elementov vseskozi hranjeno), brisanje elementov pa bi vzelo $O(m)$ časa. Ker v podatkovni zbirki Redis ni implementacije brisanja elementov glede na njihovo vrednost, bi bilo potrebno za vsak element, ki bi ga hoteli izbrisati narediti dva klica na podatkovno zbirko, kar posledično pomeni, da bi naredili vsaj enkrat več klicev na podatkovno zbirko Redis, kot v zgoraj opisanem primeru. Procesa doda-

janja in brisanja elementov iz urejene množice, ki res vzameta več časa, se odvijata na podatkovni zbirki Redis in ne blokirata samega strežnika Node.js zaradi njegove asinhrone arhitekture. Tako smo se posledično odločili za podatkovno strukturo urejene množice, saj manj obremenjuje proces strežnika Node.js.

Vse operacije preverjanje pretoka zahtevkov moramo opraviti kot celoto in jih posledično izvajamo s pomočjo transakcij. V podatkovni zbirki Redis to izvedemo s pomočjo funkcij MULTI, EXEC in WATCH. Ukaz WATCH zaklene določen element zbirke in zagotovi, da nobena druga enota medtem ne bo spremenila te vrednosti. Nato s funkcijo GET preberemo vrednost zakljenjene spremenljivke in izvedemo preverjanje prehodnosti. Na koncu v objekt MULTI zapakiramo klic funkcije SET, ki popravi vrednost spremenljivke v podatkovni zbirki. Ob klicu operacije EXEC nad prej definiranim objektom MULTI se izvedejo vse funkcije shranjene v objektu, hkrati pa se zakljenjena spremenljivka tudi sprosti.

To implementacijo smo uporabili dvakrat. Prvič za splošno kontrolo pretoka, s katero postavimo zgornjo mejo pretoka celotnega sistema in ga tako zaščitimo pred DoS napadi, in drugič za preverjanje, če uporabnikov zahtevek zadošča planu, zapisanemu v sporazumu za zagotavljanje kakovosti - SLA. Pri preverjanju zadostitvi pogojem uporabe se pred preverjanjem izvede poizvedba na podatkovno zbirko, ki vrne podatke o dovoljenem številu klicev v določenem časovnem obdobju. Na podlagi teh podatkov nato preverimo, če lahko zahtevek posredujemo naprej.

4.2.3 Spremljanje

Spremljanje uporabe je pomembno za obe strani, tako za odjemalca kot tudi ponudnika. Potrebno je premisliti potrebe obeh in koristne informacije o zahtevkih in delovanju sistema shraniti.

Omenili smo že, da smo za shranjevanje informacij o zahtevkih uporabili dokumentno podatkovno zbirko MongoDB. Dokumentne podatkovne zbirke so primerna izbira za beleženje (angl. logging), saj je struktura dinamično

prilagodljiva. Njihova slabost je slaba stopnja osveženosti med enotami v gruči, vendar to v našem primeru ni predstavljalo velike ovire, saj popolna osveženost podatkov spremljanja nima velikega vpliva.

Za hitrejše izvajanje povpraševanja po veliki količini podatkov je potrebno določiti indekse hranjenih dokumentov. Pri postavitvi podatkovne zbirke MongoDB smo definirali dva indeksa. Prvega postavimo na naslov programskega vmesnika, drugega pa na naslov končne točke, kamor je bil zahtevek poslan. Poleg omenjenih atributov iz zahtevka shranimo še povpraševalne parametre (angl. query parameters), naslov IP pošiljatelja in željen format odgovora. Pomembna podatka v odgovoru pa sta njegovo stanje in velikost sporočila. Poleg omenjenih atributov odgovora in zahtevka smo shranili še odzivni čas programskega vmesnika, ki je velikokrat najbolj zaželjen podatek pri analizi njegovega delovanja.

4.2.4 Preoblikovanje zahtevkov

Najbolj uveljavljeni strukturi podatkov v poslanih zahtevkih in prejetih odgovorih sta označevalni jezik XML in format JSON. Tako smo omogočili, da lahko v naš sistem registriramo tako programske vmesnike, ki sprejemajo vsebino v obliki označevalnega jezika XML, kot tudi sisteme, ki sprejemajo vsebino zahtevkov v formatu JSON, hkrati pa omogočamo, da uporabniške aplikacije pošiljajo in sprejemajo zahtevke obeh oblik, neodvisno od delovanja registriranega programskega vmesnika.

Ob registraciji zalednega sistema si poleg ostalih parametrov, shranimo tudi tip zahtevkov, ki jih registrirani sistem sprejema. Tako ob sprejetem zahtevku njegovo strukturo ustrezno prilagodimo ciljnemu programskemu vmesniku, hkrati pa poskrbimo tudi, da uporabniška aplikacija dobi odgovor v željeni obliki, ki jo poda v parametru glave zahtevka.

Poglavje 5

Arhitekturna zasnova prehoda

Prehod je enota, skozi katero potujejo vsi zahtevki zalednih programskih vmesnikov, zato moramo poskrbeti, da sistem ob veliki obremenitvi deluje dobro in hkrati vseskozi nemoteno deluje, saj so vsi sistemi v ozadju odvisni od njegovega delovanja. Posledično naša arhitekturna zasnova temelji na visoki stopnji dostopnosti in enostavni horizontalni skalabilnosti sistema. Hkrati smo želeli slediti granulacijskem konceptu mikrostoritev (angl. microservice), ki teži k drobitvi funkcionalnosti sistema na manjše storitve. V naslednjih podpoglavjih bomo predstavili uporabljene metode in tehnologije za dosego izpostavljenih ciljev.

5.1 Skalabilnost

Pojem skalabilnosti ponavadi označuje lastnost sistemov, da se lahko ti enostavno prilagodijo povečani uporabi, porastu količine podatkov in so dobro vzdrževani [9]. Predvsem je pomembno poudariti, da pri širjenju sistema ne pride do velikega časovnega zamika ali veliko dodatnega dela. Pojem skalabilnosti velikokrat pomešamo s pojmom učinkovitosti sistema, vendar se le-ta v našem sistemu nanaša na odzivni čas, skalabilnost pa na pretočnost sistema.

Poznamo dva načina skaliranja sistema, njuno izvajanje pa prilagajamo

strojni moči in implementaciji. Delovanje sistema lahko izboljšamo s povečanjem moči strojne opreme, na kateri se ta izvaja, ali pa z dodajanjem dodatnih enot strojne opreme, zato prvemu pravimo vertikalna, drugemu pa horizontalna skalabilnost.

5.2 Vertikalna skalabilnost

Pri vertikalnem skaliranju gre za zamenjavo stare strojne opreme z novejšo, zmogljivejšo. Sistemu ponavadi povečamo velikost pomnilnika, procesorsko moč ali prostor na trdem disku. Povečevanje strojne moči vedno prilagajamo potrebam sistema, zato ni vedno nujno posodobiti celotnega sistema. Dostikrat pa lahko naletimo na težavo nekompatibilnosti komponent in je posledično potrebno kljub temu zamenjati vse komponente. Prednost takšnega načina skaliranja je preprostost, saj ni potrebno prilagajati implementacije, ampak je sistem potrebno zgolj premestiti na drugo napravo. Aplikacije razvite v okolju Node.js imajo pri tem določene omejitve zaradi njegove enonitne arhitekturne zasnove, tako da povečevanje procesorke moči na njihovo delovanje v osnovi nima vpliva. Za izkoriščanje večjedrnih procesorjev je pri razvoju aplikacije potrebno uporabiti modul Cluster.

Velik problem vertikalnega skaliranja sta nelinearno povečavanje stroškov sistema in zgornja meja povečevanja strojne moči sistema [9]. Cena narašča veliko hitreje, kot če bi enako moč želeli sestaviti iz več manjših enot. Še večjo težavo pa predstavlja zgornja meja skaliranja. Moči strojne opreme ne moremo povečevati v nedogled, zato ima vertikalno skaliranje sistema, kljub hipotetični veliki količni denarja, zgornjo mejo, ki je ne moremo preseči.

5.3 Horizontalna skalabilnost

Princip horizontalnega skaliranja se od vertikalnega razlikuje v tem, da ne zavržemo stare strojne opreme, ampak ob njej postavimo dodatne enote. Ponavadi se v tem primeru ne uporablja zelo močne strojne opreme, ampak

več šibkejših enot sestavlja močnejšo celoto. Pomembno je poiskati pravilne komponente strojne opreme, ki jih med seboj povežemo. Zaželjeno je sistem zasnovati tako, da omogoča kombiniranje naprav z različno učinkovitostjo.

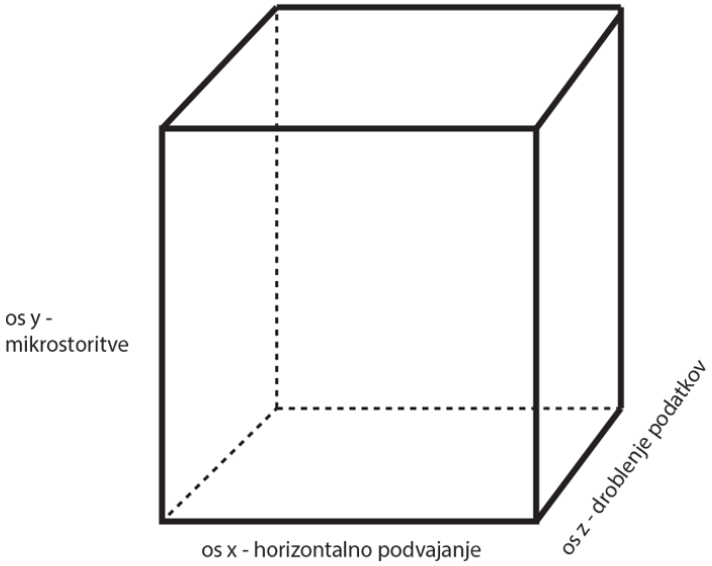
Horizontalno skaliranje je veliko cenejše od vertikalnega pristopa in ima veliko večji limit učinkovitosti. Dodatna prednost je visoka stopnja dostopnosti takšnega sistema. Če ena izmed enot sistema preneha delovati, to še ne pomeni, da se celoten sistem ustavi, ampak lahko ostale ekvivalentne enote prevzamejo delo nedelujoče komponente za čas, ko je ta nedosegljiva. Za razliko od vertikalno skaliranega principa za ta namen ni potrebno vzpostavljati pasivnih enot, ki delujejo zgolj takrat, ko aktivna enota ne deluje. Naša arhitektura meri na visoko dostopnost sistema (angl. high availability), zato je za naš sistem takšen način skaliranja bolj primeren kot pa vertikalen.

Na drugi strani je slabost takšnega pristopa povečanje administracijskih del [9]. Sistem je potrebno arhitekturno drugače zasnovati in ga prilagoditi dejstvu, da posamezna enota ne deluje individualno, ampak mora sodelovati s preostalimi. Vzporedno postavljene enote morajo delovati kot sistem brez pomnenja stanja (angl. stateless) in morajo določene spremenljivke deliti z vsemi ostalimi ekvivalenti, saj lahko njegovo delo v naslednjem trenutku prevzame sosednja enota ali pa morajo za svoje delovanje uporabljati enake spremenljivke. Vse komponente sistema je potrebno enako nastaviti, zato se potreben čas zaradi človeške narave ne povečuje linearno. Kot smo že omenili, smo mi to težavo rešili s pomočjo tehnologij Docker in Kubernetes, ki omogočata hitrejšo postavitev velikega števila enot in njihovo medsebojno konfiguracijo.

Druga težava horizontalnega skaliranja sistema je nelinearna rast učinkovitosti sistema. Čeprav linearno povečujemo njegovo velikost se dodaten čas porabi za učinkovito razporejanje obremenitev med postavljene enote. Pomembna je dobra arhitekturna zasnova in dobra ocena zgornje meje učinkovitosti sistema, saj lahko preveč naprav celo povzroči poslabšanje njegovega delovanja, zato se velikokrat odločimo za kombinacijo vertikalnega in horizontalnega širjenja sistema.

5.3.1 Načini horizontalnega skaliranja

V knjigi [1] sta avtorja predstavila tri principe horizontalnega skaliranja sistema, ki sta ga za lažjo predstavo ponazorila s pomočjo kocke, ki jo vidimo na sliki 5.1.



Slika 5.1: Načini horizontalnega skaliranja [1].

Predstavljajmo si, da postavimo n kopij enakega sistema, naloge pa med njih razporejamo s pomočjo izenačevalnika obremenitev (angl. load balancer). Takšnemu principu pravimo skaliranje po osi x. Vseh n kopij sistema deluje vzporedno, v teoriji pa vsak izmed njih prevzame $\frac{1}{n}$ nalog.

Težava takšnega pristopa k širjenju sistema je, da vsaka kopija lahko potencialno dostopa do vseh podatkov v sistemu, zato se pojavi potreba po veliki količini podatkov v predpomnilniku. Takšen pristop se ne osredotoča na velikost sistema in njegovo drobljenje. Podrobneje smo takšen način skaliranja opisali v poglavju 5.3.3.

Zelo podobno prvemu pristopu je skaliranje po osi z. V tem primeru znova

poganjamo n instanc sistema, vendar je vsak strežnik zadolžen le za določen del podatkov. Za delovanje takšnega sistema potrebujemo dodatno komponento, ki na podlagi izbranega identifikacijskega ključa določa primeren strežnik v ozadju za obdelavo posameznega zahtevka. Ta pristop se pogosto uporablja pri skaliranju podatkovnih zbirk in prinaša naslednje prednosti:

- vsak strežnik dostopa zgolj do določenega dela podatkov,
- zmanjša se količina podatkov v predpomnilniku,
- izboljša delovanje sistema v primeru uporabe transakcij.

Potrebno je upoštevati, da takšen pristop poveča kompleksnost aplikacij, saj je potrebno dodati particijsko shemo, ki lahko predstavlja težavo predvsem takrat, ko jo želimo posodobiti. Skaliranje podatkovnih zbirk smo podrobneje opisali v poglavju 5.3.2.

Na zmanjšanje kompleksnosti samega sistema in njegovo drobljenje pa se osredotoča skaliranje po osi y. Ta pristop razdrobi samo aplikacijo na več manjših enot, ki so med saboj neodvisne in skupaj sestavljajo celotno aplikacijo. Vsaka storitev ali več storitev je zadolženih za določen izoliran del funkcionalnosti aplikacije. Za takšno arhitekturo se je uveljavil izraz arhitektura mikrostoritev (angl. microservice). Gre za način drobljenja sistema na manjše smiselne enote, ki so se pojavile kot posledica prekompleknosti in zahtevnosti večjih sistemov.

5.3.2 Skaliranje podatkovnih zbirk

Tudi podatkovne zbirke lahko preobremenimo in posledično je tudi pri njih potrebno omogočiti enostavno povečevanje. Skaliranje podatkovnih zbirk je ena izmed glavnih težav skalabilnosti sistemov, saj jih veliko te možnosti ne podpira oz. vzpostavljeni sistemi ne delujejo enako dobro kot samostojne enote. Največje izzive predstavljajo potrebe po fizičnem povečevanju podatkovnih zbirk, hitrosti iskanja in posodabljanja podatkov, prilagodljivosti

veliki količini podatkov in preprečevanju nedelovanja celotnega sistema ob izpadu določenega števila strežnikov.

Pri skaliranju podatkovnih zbirk in zadoščanju zgornjim zahtevam obstajajo različni pristopi, zato bomo v nadaljevanju opisali uporabljene principe za skaliranje izbranih podatkovnih struktur. Tudi tukaj poznamo vertikalno in horizontalno rešitev širjenja sistemov. Pri vertikalnem gre za razdelitev podatkovne strukture v manjše enote (stolpce) in povečevanje velikosti ene podatkovne zbirke, pri horizontalnem pa razdelitev podatkov med več strežnikov. Uporabljenem prisotopu horizontalnega skaliranja podatkovnih zbirk pravimo drobljenje.

Drobljenje

Drobljenje je princip porazdelitve podatkov med več strežnikov, kjer so naložene enake enote podatkovne zbirke. To pomeni, da imamo izpostavljenih več ekvivalentnih tabel oz. zbirk na različnih strežnikih, med njih pa po določenem algoritmu razporejamo pripadajoče vrstice oz. dokumente.

Prednosti horizontalnega skaliranja podatkovnih zbirk so:

- razdelitev podatkov med več strežnikov in zmanjšanje količine podatkov na eni enoti,
- zmanjšanje velikosti indeksov, kar izboljša preiskovanje,
- zaradi razdeljevanja med več strežnikov se poveča moč sistema,
- podatke lahko smiselno razdelimo na podenote npr. glede na lokacijo.

Vendar pa, kot smo že prej omenili, horizontalno skaliranje podatkovnih zbirk ni trivialno in je velikokrat specifično za vsako podatkovno shemo. Pri razvoju lahko pride do določenih napak, na katere je potrebno biti pozoren, saj lahko slabo vplivajo na delovanje našega sistema. Paziti moramo na:

- večjo kompleksnost povpraševanja in posodabljanja podatkov,

- več težav pri reševanju nedelujočih strežnikov,
- večjo kompleksnost varnostnih kopij,
- več enotnih točk napak.

Podvajanje

Pri drobljenju gre za razdelitev dela med več strežnikov, od katerih noben nikoli nima vseh podatkov, ampak so porazdeljeni med vse enote. Težava se pojavi, ko eden izmed njih preneha delovati, saj noben sosednji strežnik ne vsebuje njegovih podatkov in lahko posledično preneha delovati celoten sistem. Tako bi vsak podatkovni strežnik v našem sistemu predstavljal enotno točko napake, kar se ne sklada z zastavljenim arhitekturnim konceptom.

Podvajanje (angl. replication) podatkovnih zbirk je postopek kopiranja podatkov med več strežnikov in tako zmanjša vpliv napak na določenem številu strežnikov na celoten sistem, saj v primeru nedelovanja aktivne enote delo prevzame podvojena enota, ki je do takrat v pasivnem stanju. Torej s pomočjo podvajanja povečamo zanesljivost, dostopnost in odpornost na napake.

Podatkovno podvajanje med posameznimi strežniki je ponavadi implementirano z razmerjem gospodar-suženj, kjer je gospodar strežnik, kjer se izvajajo vse operacije, ostali strežniki pa imajo status suženj in so kopije svojega gospodarja. Odvisno od implementacije se lahko na vseh strežnikih opravljajo operacije branja ali pa se vse operacije dogajajo zgolj na enem strežniku v množici podvojenih enot (angl. replication set).

Druga rešitev je večgospodaren pristop (anlg. multi-master), kjer imajo vsi strežniki enako vlogo. Največjo težavo predstavljajo transakcije in konflikti, ki jih lahko, odvisno od implementacije, rešuje podatkovna zbirka ali pa uporabnik sam. Zato smo se mi odločili za podatkovno zbirko, ki ima implementirano podvajanje gospodar-suženj, saj za ustrezno delovanje sistema potrebujemo transakcije nad podatkovno zbirko.

5.3.3 Izenačevanje obremenitev

Pri vertikalnem skaliranju za porazdelitev nalog med dodatnimi jedri skrbi operacijski sistem. Pri horizontalnem skaliranju sistema pa moramo zato poskrbeti sami oz. pred strežnike postaviti dodaten sistem, ki bo skrbel za to razporejanje. V našem primeru moramo pred gručo prehodov postaviti sistem, ki bo prejemal zahtevke in jih po določenem algoritmu razporejal med strežnike.

Najenostavnejša rešitev je kar uporaba funkcionalnosti na strežniku DNS. Naslove IPv4/IPv6 strežnikov gruče dodamo v strežnik DNS z enakim imenom strežnika (zapis tipa A/AAAA), tako da bo strežnik DNS poskrbel za razporeditev zahtevkov med registriranimi strežniki. V knjigi Building scalable web sites [9] je avtor C. Henderson izpostavil določene pomanjkljivosti takšnega pristopa:

- ne moremo implementirati svojega algoritma za razporejanje zahtevkov,
- v primeru nedelovanja enega izmed strežnikov zahtevki niso porazdeljeni med ostale delujoče, ampak so enostavno zavrnjeni,
- v primeru neustrezne konfiguracije DNS strežnika je lahko težava tudi dolgotrajnost procesa brisanja in dodajanja novih strežnikov, ki lahko vzame tudi do nekaj dni.

Opisane težave rešuje dodaten strežnik, ki ga postavimo pred gručo in skrbi za razporejanje zahtevkov. Najbolj enostavna rešitev je specifična strojna oprema. Prednost teh naprav je, da so zelo hitre in začetna konfiguracija ni zahtevna. Največjo prednost pred rešitvijo s strežniki DNS predstavlja enostavno reševanje nedelovanja strežnikov v ozadju, saj lahko njegove naloge enostavno prevzame drug delujoči strežnik. Na drugi strani imajo strojne rešitve tudi svoje slabosti, kot sta njihova visoka cena in težavnost specifične konfiguracije.

Zadnjo možnost predstavljajo programski sistemi, ki skrbijo za uravnoteženo razporejanje zahtevkov. Ti so dosti cenejši od strojnih rešitev, so pa za odtenek počasnejši in ne morejo sprejeti toliko hkratnih zahtevkov, vendar so v veliko primerih dovolj dobra alternativa. Najbolj poznane programske rešitve so Apache Server z Mod Proxy Balancer-jem, HaProxy in Nginx. Iz rezultatov raziskave [14] sklepamo, da so rešitve zelo primerljive, pozitivno pa odstopa Nginx, zato ga posledično uporabljamo tudi mi.

5.3.4 Mikrostoritve

Pri skaliranju z arhitekturo mikrostroritev ne gre za klasičen način skaliranja, kjer bi želeli sistem razširiti z namenom, da bi deloval hitreje oz. da bi lahko sprejel več nalog hkrati, ampak težimo k drobljenju kompleksnih aplikacij na manjše smiselne enote, ki so med seboj izolirane, skupaj pa predstavljajo celotno aplikacijo.

Potreba po takšni arhitekturi se je pojavila predvsem zaradi rasti kompleksnosti aplikacij, kjer so se pri uporabi monolitne arhitekture pojavile težave z daljšim razvojnim ciklom, težave pri vse pogostejših manjših spremembah, predvsem pa težave pri spremembah tehnologij, kar je kot posledico dostikrat prineslo, da so ekipe prepisale celotno aplikacijo znova.

V knjigi [12] je avtor izpostavil, da lahko z drobljenjem aplikacije na manjše storitve vsako enoto posebej naložimo na strežnik in ji lažje dodamo nove funkcionalnosti, ker je proces razvoja aplikacije bolj fleksibilen, saj ima vsaka storitev svojo poslovno logiko, uporabniški vmesnik in podatkovno zbirko. Takšna arhitektura pripomore tudi k lažjemu odpravljanju napak na strežniku, saj ob nedelovanju ene storitve ni nujno, da ne deluje celoten sistem.

Drobljenje aplikacije na manjše enote pa prinese tudi določeno dodatno kompleksnost in potrebo po boljšem razumevanju in zasnovi aplikacije, da je aplikacija ločena na smiselne enote. Komunikacija med posameznimi moduli ponavadi poteka po implicitnih povezavah, kar posledično pomeni medsebojno odvisnost določenih komponent. Hkrati se poveča čas nameščanja

aplikacije v oblak, saj moramo vse komponente naložiti in konfigurirati, da skupaj delujejo pravilno. V članku [2] so avtorji izpostavili pomen vsebnikov (angl. container) in tehnologij za orkestracijo komponent sistema, ki rešujejo ravno težavo nameščanja in usklajevanja velikega števila manjših aplikacij.

5.4 Vsebniki in Docker

Pomemben vidik enostavnega skaliranja je poenostavitev procesa nalaganja aplikacije na strežnik. Velikokrat se je težava z vzpostavitvijo sistema reševala s pomočjo programov delovnega toka (angl. workflow software) in virtualnih naprav (angl. virtual machines). Danes pa se pojavljajo okolja, ki ovijejo aplikacijo skupaj z ostalimi uporabljenimi komponentami in jo kot celoto namestijo na strežnik. Takšen ovoj aplikacije imenujemo vsebnik (angl. container). V članku [13] avtor izpostavi naslednje prednosti takšnega pristopa:

- izolacija okolja izvajanja aplikacije od operacijskega sistema,
- lahko prenosljivo okolje aplikacije,
- možnost namestitve aplikacije na veliko število strežnikov,
- možnost preprostega dinamičnega povezovanja aplikacij.

Najbolj prepoznavna predstavnika tehnologij vsebnikov sta Docker in LXC (Linux Containers). LXC je osnovni predstavnik in je tudi osnova za implementacijo rešitve Docker. LXC deluje podobno kot operacijski sistem in poleg upravljanja z vsebniki omogoča veliko njegovih funkcionalnosti. Za razliko od tega je tehnologija Docker veliko bolj strogo definirana rešitev in deluje preko svojega programskega vmesnika. Tehnologija Docker ustvari vsebnike za poganjanje aplikacij v izoliranem okolju, kopije takšnega okolja pa lahko naredimo in prenašamo s pomočjo slik (angl. images). Da lahko deluje neodvisno od operacijskega sistema, uporablja jedrne vire operacijskega sistema Linux v izolaciji in neodvisen imenski prostor, hkrati pa lahko vanj

enkapsuliramo poljubno aplikacijo. To pomeni, da je vsebnik Docker neodvisen od operacijskega sistema in neodvisen od aplikacije, ki jo poganjamo znotraj le-tega. C. Boettiger je v članku [5] iz leta 2015 izpostavil naslednje prednosti uporabe tehnologije Docker:

- enostavnejše reševanje težav z odvisnostmi,
- boljša dokumentiranost uporabljenih odvisnosti,
- enostavnejše reproduciranje okolja,
- možnost verzioniziranja okolij.

Tehnologija LXC se velikokrat uporablja pri drobljenju virov enega strežnika med več aplikacij in jih je mogoče na ta način medsebojno izolirati. Kadar pa potrebujemo tehnologijo za hitrejše nameščanje aplikacij na veliko število strežnikov pa se večinoma uporablja tehnologija Docker.

5.5 Orkestracija in Kubernetes

Kot smo opisali v poglavju 5.4, lahko s pomočjo slik tehnologije Docker, ustvarimo vsebnik poljubne aplikacije na poljubnem strežniku. Pojem orkestracije (angl. orchestration) govori o tem, kako nalagamo in usklajujemo posamezne aplikacije oz. v našem primeru vsebnike. Velikokrat se pojavi potreba po nalaganju večjih skupin enakih strežnikov, ki jih je potrebno nastaviti in uskladiti, da skupaj pravilno delujejo.

S pomočjo programskega vmesnika ogrodja Docker lahko strežnike med seboj ročno usklajujemo, vendar ta ne omogoča avtomatskega nadomeščanja nedelujočih strežnikov, hkrati pa ne zadošča potrebi po nalaganju večje skupine strežnikov in njihovo enostavno usklajevanje.

Tehnologije za orkestracijo Swarm, Fleet, Kubernetes in Mesos predstavljajo rešitve, ki se v zadnjem obdobju najpogosteje uporabljajo. V osnovi opravljajo enake naloge, vendar se v določenih konceptih razlikujejo in jih zaradi medsebojnega dopolnjevanja včasih celo srečamo skupaj v enem sistemu.

V tabeli 5.1 vidimo primerjavo omenjenih tehnologij po osnovnih kriterijih orkestracije.

Tehnologija za orkestracijo Fleet in Mesos sta nižjenivojsko usmerjeni okolji za orkestracijo in skrbita za enakomerno porazdelitev zahtevkov glede na obremenitev in razpoložljivo moč registriranih virov. Pri tehnologiji Mesos mora uporabnik sam implementirati logiko za dodeljevanje posameznih enot strojne opreme specifičnim vsebnikom (angl. scheduling), medtem ko vse ostale omenjene tehnologije to storijo same na podlagi najbolj primerne razpoložljive enote glede na potrebe posameznega vsebnika.

Tehnologija Swarm je razvita s strani razvojne ekipe tehnologije Docker in je usmerjena v orkestracijo vsebnikov Docker, dočim tehnologija Kubernetes omogoča orkestracijo poljubne implementacije vsebnikov. V članku [4] je avtor izpostavil, da je prednost tehnologije Kubernetes lažja medsebojna komunikacija in nastavitev komponent sistema, saj vse delujejo na enakem nivoju omrežja (angl. flat networking space), podpora avtomatskega nadomeščanja nedelujočih vsebnikov ter nastavitev avtomatskega skaliranja strežnikov, ki ga tehnologija Swarm ne podpira. Ravno te lastnosti so bile pomembne tudi pri naši izbiri orkestracijske tehnologije, zato smo se posledično odločili za uporabo okolja Kubernetes.

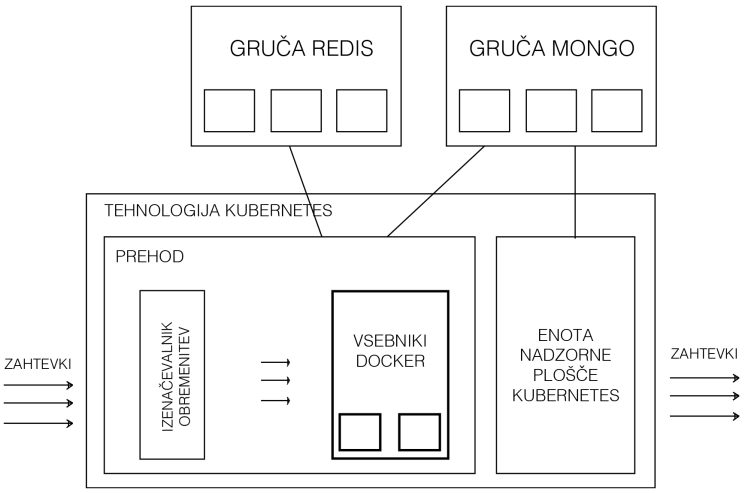
Osnovna enota v tehnologiji Kubernetes je strok (angl. pod) in predstavlja skupino vsebnikov, ki skupaj predstavljajo enoto sistema. Nato imamo instanco reprodukcijskega nadzornika (angl. replication controller), ki nadzoruje delovanje strokov, zamenjuje nedelujoče z novimi in dodaja nove stroke v primeru preobremenjenosti sistema. Na vrhu vsega pa delujejo storitve (angl. service), ki omogočajo zunanji dostop do aplikacij.

	Mesos	Fleet	Docker Swarm	Kubernetes
Visoka dostopnost	Potrebna dodatna tehnologija (npr. Zookeper)	Vzpostavljanje pasivnih enot, ki nadomeščajo nedelujoče enote	Potrebna dodatna tehnologija (npr. Zookeper)	Poskrbi za delovanje in nadomeščanje vseh enot sistema
Podpora vsebnikov	Podpira poganjanje vsebnikov Docker in Mesos	Podpira poganjanje veliko vrst vsebnikov	Podpira poganjanje vsebnikov Docker	Podpira poganjanje veliko vrst vsebnikov
Raziskovanje storitev	Domensko raziskovanje z dodatno tehnologijo	Domensko raziskovanje storitev s projektom Synapse	S pomočjo žetonov tehnologije Docker Hub ali s knjižnjico Linkv	Znotraj gruče ima vzravnano omrežje (angl. flat network)
Izenačevanje obremenitev	Lahko omogočimo z vključitvijo tehnologije HAProxy	Lahko omogočimo z vključitvijo dodatne tehnologije	Lahko omogočimo z vključitvijo dodatne tehnologije	Omogoča izenačevanje obremenitev na podlagi algoritma Round Robin
Avtomatizirano skaliranje	Ne podpira	Ne podpira	Ne podpira	Podpira
Nadzor uporabe	Omogoča definiranje uporabniških pravic	Vsak uporabnik ima vse pravice	Omogoča definiranje uporabniških pravic	Omogoča definiranje uporabniških pravic

Tabela 5.1: Tabela primerjav orkestracijskih tehnologij Mesos, Fleet, Docker Swarm in Kubernetes.

5.6 Naša arhitekturna zasnova

Predstavljene koncepte skalabilnosti smo preslikali tudi v naš sistem prehoda. Na sliki 5.2 vidimo osnovni koncept zasnovanega prehoda, v naslednjih podpoglavjih pa so podrobneje predstavljene implementacije posameznih komponent.



Slika 5.2: Skica arhitekturne zasnove našega sistema.

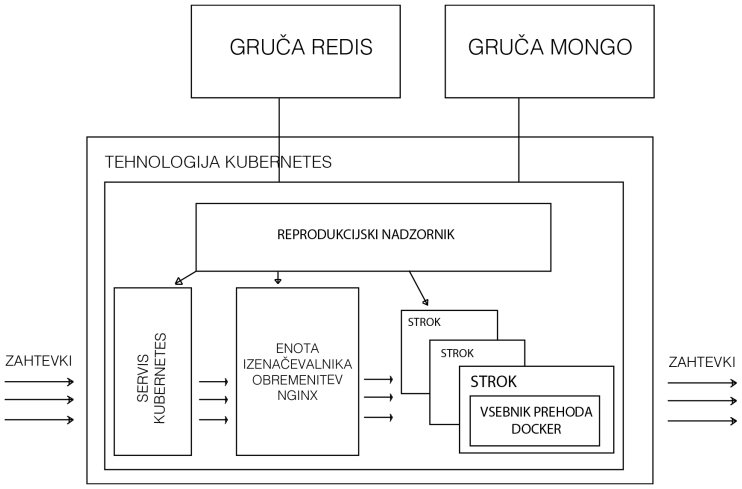
5.6.1 Skaliranje aplikacij Node.js

Predstavljen sistem mora pokrivati funkcionalnosti prehoda, ki smo jih opisali v poglavju 4.2. Odločili smo se, da bomo funkcionalnosti prehoda ločili na dve manjši storitvi. Prva storitev pokriva funkcionalnosti preusmerjanja zahtevkov, preoblikovanje zahtevkov, shranjevanje podatkov zahtevkov in nadzor pretoka ter uporabe sistema. Druga storitev pa deluje kot nadzorna plošča prehoda, kjer bomo lahko pregledali seznam prejetih zahtevkov in registrirali nov programski vmesnik. Tako smo sistem razdelili na dve manjši enoti, tako imenovani mikrostoritvi.

Aplikacijo Node.js smo razvili s pomočjo modula Cluster, ki omogoča izkoriščanje večjedrnosti procesorjev. Tako ima aplikacija en glavni proces,

ki sproži toliko suženjskih procesov, kot je število prostih jeder na procesorju. Nato ob prejetem zahtevku glavni proces razporeja prejete zahtevke med ustvarjene suženjske procese in hkrati nadzoruje njihovo delovanje ter jih po potrebi nadomešča z novimi instancami.

Za potrebe skaliranja po osi x smo obe aplikaciji ovili v vsebnik Docker. To skrajša potreben čas za izvajanje nastavitev in nalaganja sistema na strežnik. Ustvarili smo slike komponent sistema, ki smo jo nato uporabljali za podvajanje okolja na različnih strežnikih in tako omogočili skaliranje sistema s pomočjo orkestracijske tehnologije Kubernetes. Na sliki 5.3 vidimo skico našega sistema skaliranega s pomočjo tehnologije Kubernetes.



Slika 5.3: Skica našega sistema v tehnologiji Kubernetes.

Oba vsebnika sta ovita v osnovno enoto tehnologije Kubernetes strok. Za delovanje strokov, njihovo podvajanje, nadomeščanje nedelujočih enot skrbi reprodukcijski nadzornik. Z njegovo pomočjo lahko tudi ročno poljubno dodajamo in odvzemamo stroke. Tako poskrbimo za visoko dostopnost našega sistema.

Vstopno točko sistema predstavljajo servisi tehnologije Kubernetes, ki so izpostavljeni na javno dostopnem naslovu IP. Servis sam ne poskrbi za razporejanje zahtevkov med stroke, ampak izpostavi dodatno enoto izenačevalnika

obremenitev. Mi smo za to enoto izbrali sistem Nginx, ki je ovit v strok in ima svoj reprodukcijski nadzornik, ki skrbi za dobro delovanje teh enot sistema. Tudi za njegovo delovanje in podvajanje skrbi specifična enota reprodukcijskega nadzornika.

Sistemu pa lahko dodamo tudi enoto avtomatiziranega skaliranja strokov, ki ji specificiramo najmanjše in največje število enot v stroku ter ji dodamo obremenitev procesorja, ob kateri reprodukcijski nadzornik doda dodatne enote stroka.

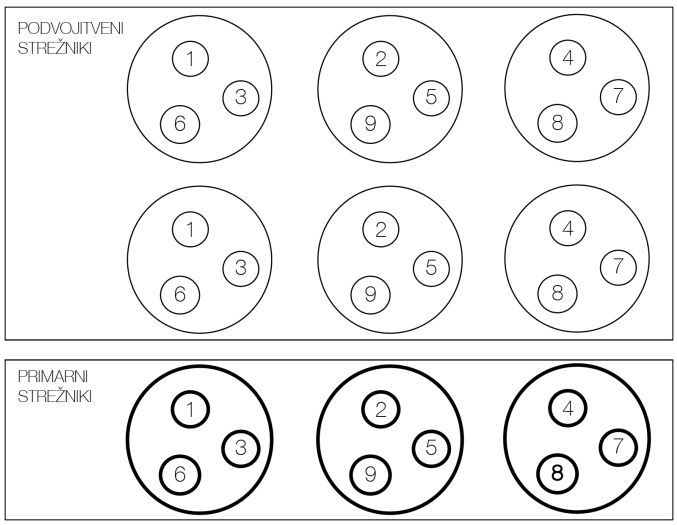
5.6.2 Skaliranje podatkovne zbirke Redis

Podatkovno zbirko Redis smo razširili s pomočjo njegovih funkcionalnosti Redis Cluster in Redis Sentinel. Cilj je bil razširiti podatke med več strežnikov, ki bodo med sabo sodelovali in delovali kot celota. Najprej smo vsak strežnik Redis posebej nastavili in ob tem omogočili vsebovanost v gruči. Poleg osnovnih vrat (angl. port), ki jih strežnik uporablja za sprejemanje klicev, smo morali sprostiti tudi dodatna vrata, ki jih strežniki Redis uporabljajo za komunikacijo med sabo. Pri tem ne pošiljajo zgolj informacij o sebi, ampak posredujejo tudi vse informacije o ostalih elementih gruče, ki jih poznajo.

Podatki v sistemu Redis Cluster so razdeljeni v tako imenovane zgoščevalne predale (angl. hash slot), katerih je 16384 in jih lahko poljubno razdelimo med strežnike. V našem sistemu imajo vse enote na voljo enako količino strojne moči, zato smo predale enakomerno porazdelili med tri strežnike. Primer takšne arhitekture vidimo na sliki 5.4, kjer omenjene strežnike predstavljajo odebeljeno očrtani krogi, znotraj pa so številke, ki predstavljajo posamezen predal. Za razvrščanje med predali skrbi zgoščevalna funkcija, katere atribut je ključ v kombinaciji ključ-vrednost, rezultat pa predstavlja predal, kamor se morajo shraniti podatki:

```
hashKey = Crc16(key) % 16384
```

Ta koncept tudi omogoča enostavno dodajanje strežnikov in premika-



Slika 5.4: Primer gruče Redis s tremi gospodarnimi strežniki, vsi pa imajo v ozadju še dve podvojeni kopiji.

nje podatkov med posameznimi strežniki. Hkrati podatkovna zbirka Redis omogoča, da to storimo, medtem ko sistem deluje, kar je zelo pomembno za skalabilnost in dostopnost sistema.

Poleg opisanih aktivnih strežnikov pa smo postavili še dodatnih šest strežnikov, od katerih bosta po dva skupaj predstavljala kopiji ene gospodarne enote. Na sliki 5.4 so ti strežniki označeni s tanjšo obrobo. Pri podatkovni zbirki Redis suženjski strežniki ne opravljajo nobene funkcije, dokler ne prevzamejo vlogo aktivnega strežnika.

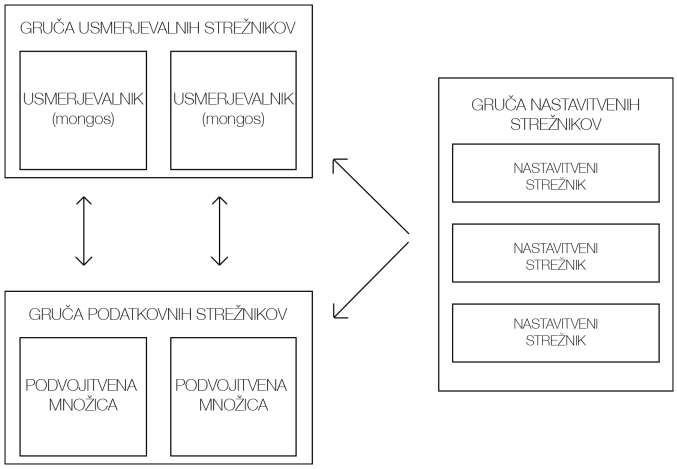
Uporabili smo tudi funkcionalnost Redis Sentinel in vzporedno postavili še dodatne tri strežnike, ki pa za razliko od prej predstavljenih strežnikov ne hranijo podatkov, ampak skrbijo za njihovo nadomeščanje v primeru nedelovanja in obveščanje skrbnikov o stanju vseh strežnikov gruče.

5.6.3 Skaliranje podatkovne baze MongoDB

Tako kot pri podatkovni zbirki Redis smo vzpostavili tudi gručo podatkovne zbirke MongoDB. Ta se od gruče podatkovne zbirke Redis razlikuje predvsem

v tem, da za razporejanje podatkov med strežnike ne skrbi uporabnik, ampak postavimo dodatne strežnike, ki opravljajo to delo.

Najprej smo poskrbeli za podvajanje strežnikov MongoDB. Postavili smo štiri enote strežnikov MongoDB in jih nastavili tako, da po dve in dve pripadajo posamezni podvojitveni množici. Ti sta dostopni preko obeh vsebovanih strežnikov in preddoločenega imena. Nato smo vzpostavili dodatne tri strežnike, ki jih sistem MongoDB potrebuje za nadzor delovanja gruče in jih imenujemo nastavitveni strežniki. Nazadnje smo nad nastavitvenimi strežniki postavili še zadnja dva, ki predstavljata vstopno točko gruče in ju imenujemo usmerjevalna strežnika. Za vzpostavitev celotnega sistema smo morali usmerjevalnima strežnikoma nastaviti, kateri so podatkovni strežniki gruče. Pri tem smo definirali, naj podatke med njimi razporejata na podlagi polja z naslovom programskega vmesnika. Takšna struktura je prikazana na sliki 5.5.



Slika 5.5: Skaliranje podatkovne baze MongoDB

Pri tem bi poudarili dve značilnosti izpostavljenega sistema. Pri podatkovni zbirki MongoDB suženjski strežniki v sistemu ne sodelujejo popolnoma pasivno, ampak lahko na njih vseskozi izvajamo operacije branja. Druga pomembna lastnost gruče sistema MongoDB pa je, da posreduje povratne

informacije v obliki odgovora uporabniku takoj, ko je ta uspešno zaključena na glavnem strežniku. Posledica tega je, da se lahko zaradi napake na glavnem strežniku izgubijo določeni podatki, ki še niso bili podvojeni. Sistem je mogoče tudi drugače nastaviti in počakati, da se podatki zapišejo tudi na pasivne strežnike, vendar se mi za tak pristop nismo odločili, saj menimo, da izguba zgolj majhnega števila podatkov ne bi drastično spremenila informacij o delovanju posameznega programskega vmesnika.

Poglavje 6

Vrednotenje zmogljivosti prehoda

V tem poglavju bomo predstavili rezultate opravljenih testiranj, s katerimi smo preverili delovanje implementiranega sistema. Z opravljenimi testi smo naslovili predvsem vprašanja o dodatnem času, ki ga zahtevek porabi zaradi potovanja skozi prehod, možnost da bi implementiran prehod predstavljal ozko grlo sistema in skalabilnost ter prilagajanje sistema večjim obremenitvam. Na začetku bomo naredili tehničen pregled testnega okolja in načina testiranja, zatem pa bomo predstavili in razložili pridobljene rezultate.

6.1 Testno okolje

Vzpostavili smo testno okolje za izvajanje testov učinkovitosti (angl. preformance testing) in obremenitvenih testov (angl. load testing). Testno okolje smo postavili v oblaku s pomočjo opisane tehnologije Kubernetes. Sistem je postavljen na štirih ekvivalentnih strežnikih ponudnika oblačnih storitev Google Cloud Platform [21]. Na strežnikih je naložen operacijski sistem CentOS, ki ima na voljo 7.5 GB pomnilniškega prostora in dvojederni 2.6GHz procesor Intel Xeon E5. Tudi gruči podatkovnih zbirk Redis in MongoDB sta naloženi na enakih strežnikih.

Za potrebe izvajanja testov smo v ozadju vzpostavili še tri dodatne strežnike Node.js, ki simulirajo delovanje programskih vmesnikov. Vsi se odzivajo enako hitro in so postavljeni na podobnih lokacijah na strežnikih ponudnika oblačnih storitev Digital Ocean [22]. Teh strežnikov z razlogom nismo postavili v enako okolje kot preostale enote prehoda, saj bi tako odvzeli določen del strojne moči našemu sistemu, hkrati pa smo želeli kar najbolje simulirati stanje v realnem okolju.

6.2 Testno orodje

Za testno orodje smo izbrali odprtokodno rešitev Locust [24]. V primerjavi s testnim orodjem JMeter [23], ki ponuja veliko več funkcionalnosti, je veliko lažje razširljiv oz. skalabilen. Mi smo potrebovali skalabilno rešitev, saj iz lokalnega okolja nismo uspeli poslati dovolj zahtevkov.

Tako smo orodje Locust s pomočjo tehnologije Kubernetes postavili na štirih strežnikih ponudnikov oblačnih storitev Google Cloud Platform. Tudi ti strežniki imajo naložen operacijski sistem CentOS, ki pa ima na voljo 3.5 GB pomnilniškega prostora in 2.6GHz procesor Intel Xeon E5. Strežniki se nahajajo na enaki lokaciji kot strežniki našega sistema, saj smo tako dobili natančnejše rezultate o samem delovanju našega sistema in se izognili morebitnim težavam v omrežju.

Obdelavo in grafičen prikaz pridobljenih rezultatov smo kasneje naredili s pomočjo orodja GNU Octave [25].

6.3 Razlaga izmerjenih rezultatov

V prvem delu tega razdelka bomo predstavili rezultate testov učinkovitosti, kjer smo pridobili podatke o hitrosti delovanja uporabljenih komponent in celotnega sistema. V drugem podpoglavju pa bomo opisali rezultate obremenitvenih testov, kjer smo preizkusili zmožnosti postavljenega skalabilnega sistema.

6.3.1 Testi učinkovitosti

Pri testih učinkovitosti je bilo pomembno, da strežnikov nismo preobremenili, ampak da smo v dolgem časovnem intervalu poslali zadostno količino zahtevkov. Za najboljši približek delovanja sistema smo vzeli povprečen odzivni čas in mediano odzivnih časov. Pri testiranju učinkovitosti smo vzporedno vzpostavili 100 niti, kjer je vsaka nit vsako sekundo poslala nov zahtevek. Pri vseh testiranjih smo poslali približno 6000 zahtevkov.

Pri izvajanju vsakega testa smo pošiljali zahtevke na tri različne strežnike v ozadju, kar lahko prepoznamo v končnici posameznega zahtevka. Vsi strežniki za obdelavo posameznega zahtevka porabijo 20 milisekund.

Rezultate posameznega testiranja smo predstavili v tabeli, kjer vsaka vrstica predstavlja pošiljanje zahtevkov na posamezen programski vmesnik v ozadju, zadnja vrstica pa predstavlja seštevek vseh rezultatov. V prvem stolpcu je naziv programskega vmesnika, v drugem stolpcu imamo število poslanih zahtevkov, nato pa po vrsti sledijo mediana odzivnih časov poslanih zahtevkov, povprečen odzivni čas zahtevkov, minimalen odzivni čas zahtevkov, maksimalen odzivni čas zahtevkov in število prejetih zahtevkov na sekundo.

Ime	# zahtevkov	Mediana [ms]	Povprečje [ms]	Min [ms]	Max [ms]	# zahtevkov [1/s]
/api1	2043	31	31	28	238	32.85
/api2	2068	31	31	28	52	33.14
/api3	2042	31	31	28	68	32.81
Total	6153	31	31	28	238	98.80

Tabela 6.1: Tabela testov učinkovitosti strežnikov, ki simulirajo delovanje programskih vmesnikov (Test 1).

Pri testiranju Test 1 smo izmerili odzivne čase strežnikov Node.js, ki simulirajo delovanje programskih vmesnikov. Pridobili smo rezultate o njihovih odzivnih časih, ki so prikazani v tabeli 6.1. Hkrati smo na te strežnike poslali velike količine podatkov in ugotovili, da tudi ob veliki obremenitvi delujejo hitro in tako ne bodo vplivali na pridobljene rezultate.

Pri naslednjih testih smo vzpostavili implementiran prehod z različno količino funkcionalnosti in tako preverili, koliko se odzivni čas poveča kot posledica dodatnih operacij. Pri vseh testih smo prehod naložili zgolj skozi en strok tehnologije Kubernetes, kar pomeni, da je bil naložen le na enem izmed štirih vzpostavljenih strežnikov.

Pri testiranju Test 2 smo preverili, koliko časa pridobimo, če zahtevek pošljemo na prehod brez funkcionalnosti, ki zahtevek prejme in ga takoj preusmeri na določen uporabniški vmesnik v ozadju.

Ime	# zahtevkov	Mediana [ms]	Povprečje [ms]	Min [ms]	Max [ms]	# zahtevkov [1/s]
/api1	1784	41	42	38	99	32.64
/api2	1796	41	43	38	96	32.21
/api3	1818	41	42	38	105	32.93
Total	5398	41	42	38	105	97.78

Tabela 6.2: Tabela testov učinkovitosti prehoda brez dodatnih funkcionalnosti (Test 2).

Rezultate pošiljanja zahtevkov na sistem, ki zgolj posreduje zahtevke na programske vmesnike, vidimo v tabeli 6.2. Ugotovili smo, da se je povprečen odzivni čas povečal za 10 milisekund kot posledica podaljšane poti zahtevkov. Vidimo, da se je minimalni odzivni čas zahtevka tudi povečal za 10 milisekund, medtem ko se je maksimalen odzivni čas povečal za 30 milisekund. To je posledica podaljšane poti, ki jo opravijo zahtevki in so posledično bolj izpostavljeni napakam in obremenitvam v omrežju.

Ime	# zahtevkov	Mediana [ms]	Povprečje [ms]	Min [ms]	Max [ms]	# zahtevkov [1/s]
/api1	1825	48	49	40	228	31.90
/api2	1863	47	51	40	231	31.07
/api3	1873	48	50	40	208	31.78
Total	5561	47	50	40	238	94.75

Tabela 6.3: Tabela testov učinkovitosti prehoda s funkcionalnostma preusmerjanja in spremljanja (Test 3).

V tabeli 6.3 vidimo rezultate testiranja Test 3, kjer smo prehodu dodali

funkcionalnosti preusmerjanja in spremljanja zahtevkov. To pomeni, da smo sistemu dodali operacije branja in pisanja v podatkovno zbirko MongoDB. Iz pridobljenih rezultatov je razvidno, da se je povprečen odzivni čas, kot posledica dodatnih funkcionalnosti, povečal za 6 milisekund. Pri tem se je minimalen odzivni čas povečal za 2 milisekundi, maksimalen pa za približno dodatnih 100 milisekund.

Pri testiranju učinkovitosti Test 4 smo dodali še funkcionalnost nadzora pretoka in uporabe, kar pomeni, da smo vzpostavili celoten implementiran sistem. Tako smo dodali operacije branja in pisanja v podatkovno zbirko Redis. Iz rezultatov v tabeli 6.4 lahko razberemo, da se je povprečen odzivni čas povečal še za približno 5 milisekund. Minimalen odzivni čas se je od testiranja Test 3 povečal za 2 milisekundi, malo pa se je zmanjšal odzivni čas zahtevka z maksimalnim odzivnim časom.

Ime	# zahtevkov	Mediana [ms]	Povprečje [ms]	Min [ms]	Max [ms]	# zahtevkov [1/s]
/api1	2032	51	54	42	166	31.68
/api2	2102	50	54	43	201	32.06
/api3	2072	51	55	42	178	32.46
Total	6206	51	55	42	201	96.20

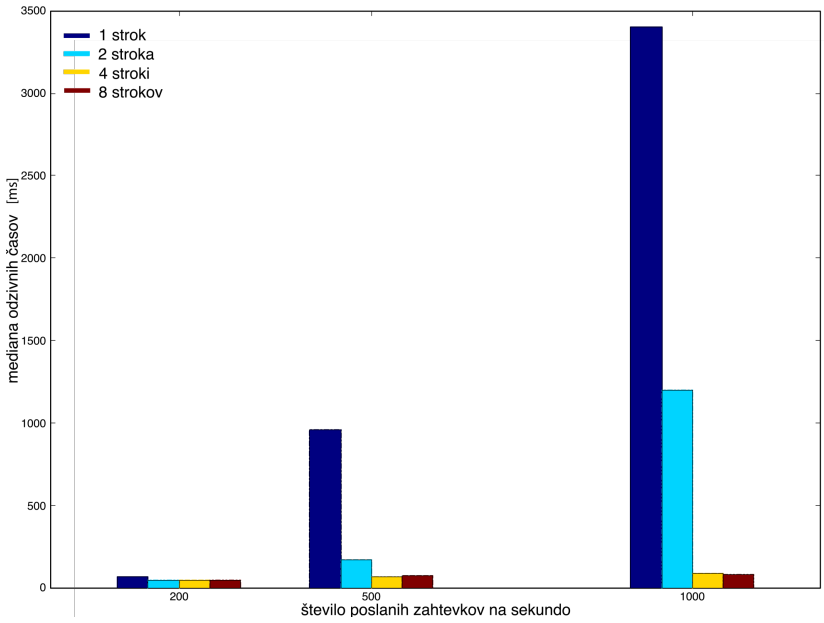
Tabela 6.4: Tabela testov učinkovitosti prehoda s funkcionalnostmi preusmerjanja, spremljanja in nadzora pretoka ter uporabe (Test 4).

Vse funkcionalnosti skupaj so prispevale malo več kot 10 milisekund k posameznemu zahtevku. Ta rezultat lahko primerjamo z rezultati testiranja podobne komponente Kong [29], ki jo razvija ekipa podjetja Mashable. Avtorji komponente so opravili teste učinkovitosti [30] in ugotovili, da njihova komponenta v povprečju za posamezen zahtevek potrebuje približno 10 milisekund. Testiranja so opravili na strežnikih ponudnika Amazon z dvojedrnimi procesorji in 4GB pomnilniškega prostora. Če se zavedamo, da je njihovo testno okolje zelo primerljivo z našim in da je to komponenta, ki jo uporabniki uporabljajo v produkciji, lahko ugotovimo, da naš sistem deluje zadovoljivo hitro.

6.3.2 Obremenitveni testi

Z obremenitvenimi testi smo preverili, kako se naš sistem odziva pri večjih obremenitvah in ali je naš sistem dovolj skalabilen, da se tem obremenitvam tudi uspešno prilagodi.

Pri izvajanju obremenitvenih testov smo pri vseh testih ohranili enako okolje, pri tem pa smo spreminjali število strokov (angl. pod) tehnologije Kubernetes, preko katerih je vzpostavljen implementiran prehod. Teste smo opravili pri obremenitvah 200, 500 in 1000 zahtevkov na sekundo ob vzpostavitvi 1, 2, 4, ali 8 strokov našega sistema. Med testiranjem nismo spreminjali ali skalirali nobene druge komponente sistema. Pri vsakem testu smo skupaj poslali približno 50.000 zahtevkov, pred začetkom pošiljanja pa smo število zahtevkov na sekundo počasi stopnjevali, tako da sistem ni bil v trenutku polno obremenjen.



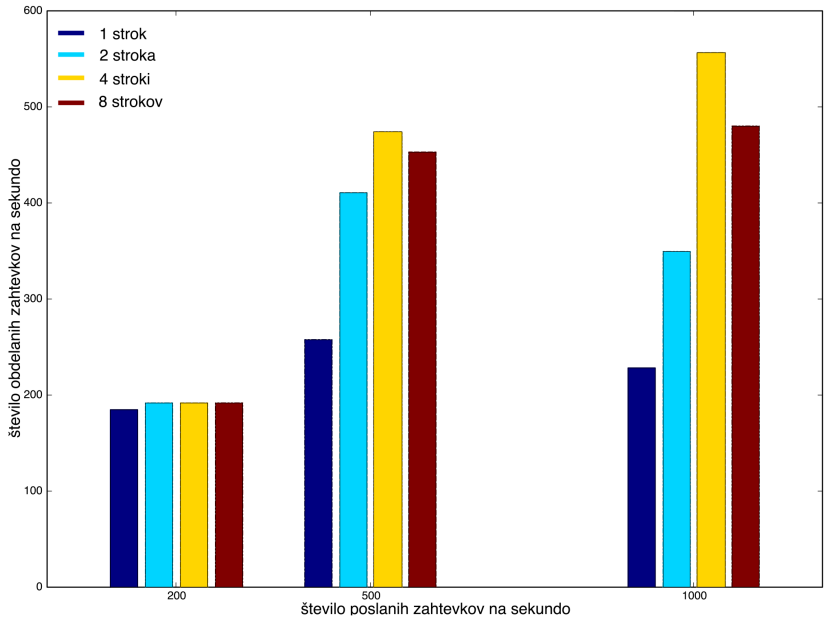
Slika 6.1: Stolpčni diagram median odzivnih časov sistema ob različnih obremenitvah in različnem številu strokov sistema.

Na sliki 6.1 vidimo stolpčni diagram mediane odzivnih časov posameznega

testiranja. Vidimo, da vsi sistemi zadovoljivo dobro delujejo pri obremenitvi 200 zahtevkov na sekundo.

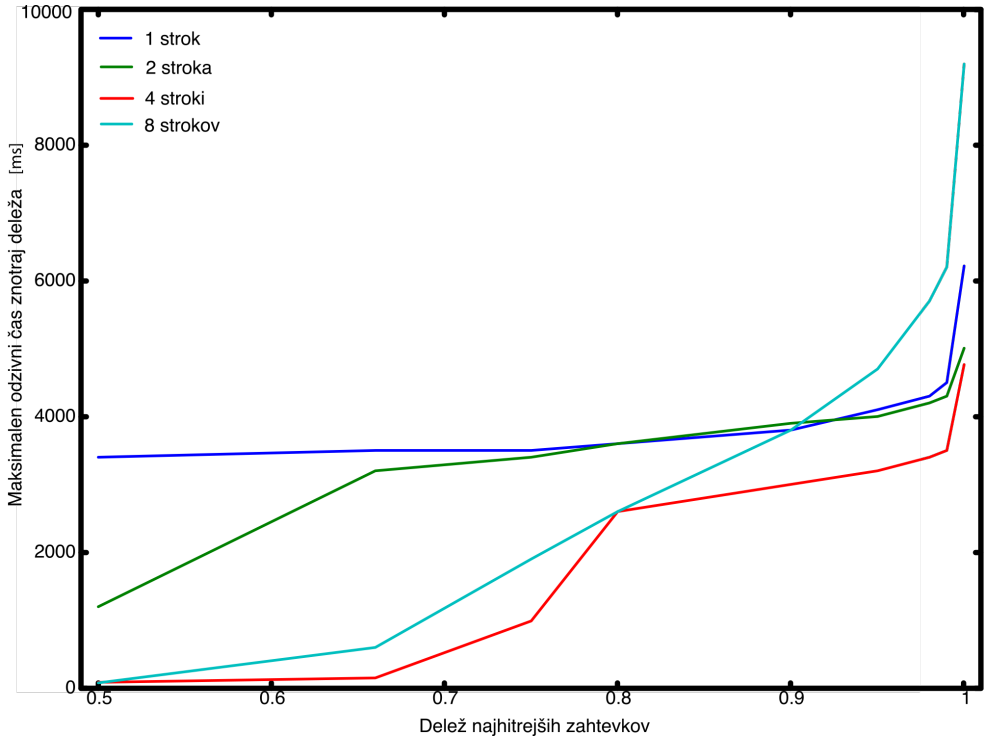
Večja razlika pa se pojavi pri obremenitvi sistema s 500 zahtevki na sekundo. Opazimo lahko, da je se je močno poslabšalo delovanje sistema, ki deluje znotraj enega stroka tehnologije Kubernetes. V tem primeru odzivni čas močno naraste in sistem je potrebno prilagoditi tej obremenitvi. Opazimo, da povečanje sistema na dva stroka skoraj popolnoma odpravi težavo, sistem pa znova deluje normalno ob vzpostavitvi s štirimi in osmimi stroki. Hkrati vidimo, da sistem s štirimi stroki obdela več zahtevkov v enakem časovnem obdobju kot sistem z osmimi stroki. To anomalijo bomo razložili v nadaljevanju poglavja.

Podobno obnašanje lahko opazimo tudi pri testiranju s 1000 zahtevki na sekundo. Vidimo, da sistem ob tej obremenitvi ne deluje dobro niti, ko je ta vzpostavljen skozi dva stroka, še vedno pa smo dobili zadovoljive rezultate sistemov na štirih in osmih strokih.



Slika 6.2: Stolpčni diagram števila obdelanih zahtevkov na sekundo ob različnih obremenitvah in različnem številu postavljenih strokov sistema.

Seveda pa mediana odzivnih časov ni edino merilo delovanja sistema ob večji obremenitvi. Na sliki 6.2 vidimo stolpčni diagram števila obdelanih zahtevkov na sekundo ob posamezni obremenitvi. Tudi tukaj lahko ugotovimo, da vsi sistemi delujejo brez težav pri obremenitvi 200 zahtevkov na sekundo. Enako kot pri mediani odzivnih časov lahko tudi tukaj ugotovimo da sistema, ki sta izpostavljena z enim ali dvema strokoma, naletita na težave pri obremenitvah s 500 in 1000 zahtevki.



Slika 6.3: Graf najnižjih odzivnih časov zahtevkov glede na določen delež vseh zahtevkov.

Več pozornosti pa lahko tukaj namenimo izpostavitvi sistema s štirimi in z osmimi stroki, kjer lahko izpostavimo dodatno ugotovitev. Sistem, ki je izpostavljen s štirimi stroki tehnologije Kubernetes, deluje bolje kot sistem z osmimi stroki. To lahko opazimo tudi na grafu najnižjih odzivnih časov zahtevkov glede na določen delež vseh zahtevkov, ki je prikazan na sliki 6.3. Na

osi x je predstavljena količina opazovanih zahtevkov. Za izbran delež izmed vseh zahtevkov izberemo tiste z najkrajšim odzivnim časom. Na osi y grafa pa je predstavljen najdaljši odzivni čas zahtevka znotraj opazovanega deleža. Vidimo, da je najbolje deloval sistem izpostavljen s štirimi stroki, malo slabše pa sistem z osmimi stroki, ki pa je pri določenem malem deležu zahtevkov imel celo slabše odzivne čase kot sistema z enim in dvema strokoma.

Razloge za takšne rezultate lahko iščemo v število naprav, ki jih poganjamo v ozadju in številu strokov, ki jih poganjamo na enem strežniku. Pri izpostavitvi sistema preko osmih strokov imamo na vseh strežnikih dva stroka in tako tudi dve aplikaciji Node.js, ki si delita vire strežnika. To težavo izpostavijo tudi avtorji članka [6], ki so kot razlog navedli težave v predpomnilniku, ki ga operacijski sistem ob menjavi procesov pobriše in tako procesa, ki se menjujeta v delovanju, ne moreta v popolnosti izkoristiti njegove hitrosti in posledično celoten sistem deluje za odtenek slabše.

Z rezultati obremenitvenih testov smo ugotovili, da smo razvili skalabilno rešitev prehoda v ogrodju Node.js, ki se lahko s pomočjo vsebnikov Docker in tehnologije Kubernetes uspešno prilagaja povečani obremenitvi.

Poglavje 7

Zaključek

Področje programskih vmesnikov danes ponuja kopico rešitev, njihov obseg pa se bo z naraščanjem količine podatkov samo še bolj razširil. Zaradi potrebe po nadzoru in spremljanju delovanja programskih vmesnikov se je razvilo področje upravljanja programskih vmesnikov. Med raziskovanjem tematike diplomske naloge smo spoznali, da je pomembnost šibke sklopljenosti, enostavne skalabilnosti in visoke odzivnosti zelo visoka na področju programskih vmesnikov ravno zaradi hitre rasti števila naprav in aplikacij, ki te podatke in programsko logiko uporabljajo. Programski vmesniki so zaradi svoje pomembnosti in uporabe postali najbolj obremenjeni sistemi v spletnem okolju, hkrati pa je od njihovega delovanja odvisno veliko število aplikacij, ki izpostavljene podatke oz. programsko logiko uporabljajo. Posledično se je pojavila potreba po hitri obdelavi velike količine zahtevkov in zmanjšanju možnosti nedelovanja sistema.

V diplomskem delu smo zasnovali elastično arhitekturo prehoda za upravljanje programskih vmesnikov. Poskrbeli smo za skalabilnost in visoko dostopnost tako implementiranega prehoda, kot tudi uporabljenih podatkovnih zbirk Redis in MongoDB. Razlog za razvoj takšnega okolja je bil izostanek pomembnosti elastične arhitekture določenih ponudnikov sistemov za upravljanje programskih vmesnikov, kot sta Apiman [27] in Informatica [28].

Ogrodje Node.js se je v računalniškem okolju uveljavilo zaradi dogod-

kovno usmerjene arhitekture, ki omogoča sproščanje strežnika med čakanjem na odgovor zahtevka in možnostjo velike količine hkratnih povezav. Tudi mi smo zaradi teh dveh lastnosti izbrali ravno ogrodje Node.js, saj implementacija prehoda ne zahteva veliko procesorskega časa in moči, ampak večino časa komunicira in kombinira informacije, pridobljene iz drugih okolij.

Še en pomemben vidik razvoja takšnega sistema je enostavna namestitev sistema in možnost hitre prilagoditve le-tega večjim obremenitvam. Posledično smo prehod implementirali po načelih arhitekture mikrostoritev, te zapakirali v vsebnike Docker in jih na strežnike namestili s pomočjo orkestracijske tehnologije Kubernetes. V diplomskem delu smo pokazali, da se lahko implementiran prehod enostavno prilagaja večjim obremenitvam. Pri obremenitvenem testiranju smo preobremenili sistema znotraj enega in dveh strokov tehnologije Kubernetes, hkrati pa smo pokazali, da sistem, izpostavljen znotraj štirih strokov, nemoteno deluje pri obremenitvi 1000 zahtevkov na sekundo. Sistem smo zasnovali tako, da ne deluje ves čas na celotni zbirki zakupljene strojne opreme, ampak se lahko s pomočjo avtomatiziranega skaliranja razmnožuje na dodatne strežnike, ob tem pa nemoteno sprejema zahtevke.

V primeru uporabe prehoda, ki ima pomembno vlogo pri nadzoru in upravljanju programskih vmesnikov, pa se pojavi tudi negativen vidik zaradi podaljšanja odzivnega časa zahtevkov, ki potujejo skozi prehod. Posledično smo bili pri razvoju našega sistema pozorni na to problematiko in smo s testi učinkovitosti ugotovili, da se mediana in povprečni odzivni čas zahtevkov zaradi podaljšane poti skozi prehod podaljšata za približno 10 milisekund. Posledično smo z referenčnimi primeri ugotovili, da lahko hitrost našega sistema primerjamo s podobnimi produkcijskimi tehnologijami, ki so bila testirana na strežnikih primerljive zmogljivosti. Seveda ne smemo pozabiti na izboljšave sistema. Predvsem je možno implementiran prehod funkcionalno razširiti. Na primer z modulom za avtentikacijo uporabnikov.

Literatura

[1] M. L. Abbott, M. T. Fisher, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*, Pearson Education, 2009.

[2] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, M. Steinder, "Performance Evaluation of Microservices Architectures using Containers", v zborniku *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on Network Computing and Applications*, str. 27–34, 2015.

[3] V. Alagarasan, R. C. Huacarpuma, S. R. Lima, W. Dean, "API Management Introduction and Principles", Forrester research, 2015.

[4] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes", *IEEE Cloud Computing*, št. 3, str. 81-84, 2014.

[5] C. Boettiger, "An introduction to Docker for reproducible research", *ACM SIGOPS Operating Systems Review*, št. 49, zv. 1, str. 71-79, 2015.

[6] M. Harkema, B.M.M. Gijsen, R. D. van der Mei, L. J. M. Nieuwenhuis, "Performance comparison of middleware threading strategies", v zborniku *Proc. of Int. Symp. Performance Evaluation of Computer and Communication Systems*, str. 727-732, 2004.

[7] R. Cattell, "Scalable SQL and NoSQL data stores", *ACM SIGMOD Record*, št. 39, zv. 4, str. 12-27, 2015.

[8] D. Crockford, *JavaScript: The Good Parts*, O'Reilly Media, Inc., 2008, pogl. 3.

[9] C. Henderson, *Building scalable web sites*, O'Reilly Media, Inc., 2006, pogl. 9.

[10] K. Ma, R. Sun, "Introducing websocket-based real-time monitoring system for remote intelligent buildings", *International Journal of Distributed Sensor Networks*, št. 2013, 2013.

[11] O. Michels, C. Loveless, "Methods for analyzing, limiting, and enhancing access to an internet API, web service, and data", US Patent 9 027 039, 05, 2015. Dostopno na: https://www.google.com/patents/US9027039.

[12] S. Newman, *Building Microservices*, O'Reilly Media, Inc., 2015, pogl. 1, 2.

[13] P. Claus, "Containerization and the PaaS cloud", *IEEE Cloud Computing*, št. 3, str. 24-31, 2015.

[14] A. Piórkowski, A. Kempny, A. Hajduk, J. Strzelczyk, "Load balancing for heterogeneous web servers", v zborniku *17th Conference on Computer Networks*, Ustroń, Poland, jun. 2010, str. 189-198.

[15] S. Tilkov, S. Vinoski, "Node. js: Using JavaScript to build high-performance network programs", *IEEE Internet Computing*, zv. 6, str. 80-83, 2010.

[16] M. Welsh, D. Culler, E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services", *ACM SIGOPS Operating Systems Review*, št. 35, zv. 5, str. 230-243, 2001.

[17] StrongLoop API Gateway. [Online]. Dosegljivo: http://docs.strongloop.com/display/LGW/StrongLoop+API+Gateway. [Dostopano: 5. 6. 2016].

[18] KPIs for APIs: developer experience can make or break your API. [Online]. Dosegljivo: http://www.programmableweb.com/news/kpis-apis-developer-experience-can-make-or-break-your-api/analysis/2014/10/21. [Dostopano: 5. 6. 2016].

[19] Request throttling – a proactive approach. [Online]. Dosegljivo: https://enterprisegateway.wordpress.com/2011/08/04/request-throttling-a-proactive-approach/. [Dostopano: 6. 6. 2016].

[20] Improve API Gateway Throttling. [Online]. Dosegljivo: http://ccoetech.ebay.com/improve-api-gateway-throttling. [Dostopano: 6. 6. 2016].

[21] Google Cloud Platform. [Online]. Dosegljivo: https://cloud.google.com/. [Dostopano: 10. 6. 2016].

[22] Digital Ocean. [Online]. Dosegljivo: https://www.digitalocean.com/. [Dostopano: 11. 6. 2016].

[23] JMeter. [Online]. Dosegljivo: https://jmeter.apache.org/usermanual/index.html. [Dostopano: 11. 6. 2016].

[24] Locust. [Online]. Dosegljivo: http://locust.io. [Dostopano: 11. 6. 2016].

[25] GNU Octave. [Online]. Dosegljivo: https://www.gnu.org/software/octave. [Dostopano: 11. 6. 2016].

[26] API Providers Guide - API Management. [Online]. Dosegljivo: http://pages.3scale.net/rs/3scale/images/KL-api-managment-providers-guide.html?mkt_tok=3RkMMJWWfF9wsRonuqTNZKXonjHpfsX56OgvWKK%2BlMI%2F0ER3fOvrPUfGjI4DRMVm. [Dostopano: 19. 7. 2016].

[27] Apiman. [Online]. Dosegljivo:
http://www.apiman.io/latest/roadmap.html. [Dostopano: 14. 7. 2016].

[28] IronCloud API management platform. [Online]. Dosegljivo:
https://www.strikeiron.com/about-the-ironcloud-platform/. [Dostopano: 15. 6. 2016].

[29] Kong. [Online]. Dosegljivo:
https://getkong.org. [Dostopano: 20. 8. 2016].

[30] Kong Performance Benchmark. [Online]. Dosegljivo:
https://getkong.org/about/benchmark/. [Dostopano: 20. 8. 2016].